MUNIX Pascal Package

PCS
Pfaelzer-Wald-Str. 36
8000 Muenchen 90

This manual contains command descriptions in the style of MUNIX Programmers' Manual Volume 1 and supplementary documents.

1. MUNIX Pascal Package - Summary

2. Pascal 68000 User's Guide Version 1.1

3. Learn Snobol

4. PBUG ein interaktiver bildschirmorientierter Debugger

5. Command Descriptions

October 29, 1982

MUNIX Pascal Package V1.1 - Summary

PCS
Pfaelzer-Wald-Str. 36
8000 Muenchen 90

The MUNIX Pascal Package includes not only the PASCAL programming language, but also the SNOBOL language and additional utilities.

The screen oriented utilities of this package work on a large number of display terminals. New terminals are easily driven after editing a terminal description file.

1. Languages

PC — Compile and/or link programs in the PASCAL 68000 language.
PASCAL 68000 is an extended implementation of the PASCAL language. Specifically PASCAL 68000 complies almost completely with the requirements of the ISO standard proposal for PASCAL. Some of the features of PASCAL 68000 are

- General purpose language
- Block oriented language
- Strong type checking
- Variety of data structures:
  simple types, arrays, records,
  sets, files, pointers
- Various control statements
- Predefined procedures and functions
- Seperate compilation of modules
- Import and export of variables, procedures and functions

November 5, 1982

MUNIX PASCAL PACKAGE

PCS GmbH
Periphere Computer Systeme
[illegible]ilzer-Wald-Straße 36
[illegible]8000 München 90
Telefon (0 89) 68 10 21
Telex 5 23 271

- Linking C, FORTRAN or assembler modules to PASCAL 68000 modules

SNO — Interpret programs in the SNOBOL language.

- SNOBOL has some of the best features of Basic and Lisp: It is interactive, has 'rubber memory' for strings, lists and associative tables, and finally it is easy to learn.

- SNOBOL is qualified for the following applications:
  text editing jobs
  small interactive data bases
  small translators: mini-languages, macros...

- Preprocessor snif

2. Text Processing

PED — Interactive screen oriented editor . Most of the facilities of the ED line editor with the extension of free cursos movement on the screen and the use of function keys. The mnemonic assignement of commands to keys makes the editor command set easy to use and to remember.

- Add, delete, change, copy lines.

- Split, concatenate lines.

- Find lines by number or pattern.

- Manage previous defined rectangles.

- Switch to another file.

- Define abbreviations.

- Escape to Shell during editing.

PAGE — Interactive display function for text files.

- Examine a file on screen sequential by use of function keys.

- Choose an arbitrary page of the file by cursor positioning.

3. Program Development Tools

PBUG — Screen oriented source level run time debugger for C, PASCAL and FORTRAN programs.

- PBUG splits the screen into several windows for
  - command menu
  - display of variables or symbolnames
  - memory dump
  - error messages
  - I/O to resp. from the debugged program
- The easy-to-learn command interface allows the user to
  - set and reset breakpoints
  - display the source code of the debugged program
  - list the contents of variables and memory
  - start, continue and kill the program
  - get information of the available commands

XREF — create a cross-reference listing from a C or Pascal program.

Pascal 68000 User's Guide
Version 1.1
June 82

ABSTRACT

The Pascal 68000 User's Guide is intended for use in developing new Pascal programs, and in compiling and executing existing Pascal programs on 68000-UNIX† systems. This manual gives also some insight into the Pascal 68000 System structure, its components and its behaviour.

Pascal 68000 is an extended implementation of the Pascal language. Specifically Pascal 68000 complies almost completely with the requirements of the ISO standard proposal for Pascal.

This manual is designed for programmers who have a working knowledge of Pascal. Detailed knowledge of 68000 UNIX is helpful but not essential.

†UNIX is a Trademark of Bell Laboratories.

TABLE OF CONTENTS

11. Introduction

The Pascal 68000 User's Guide is intended for use in developing new Pascal programs and in compiling and executing existing Pascal programs on 68000 UNIX systems. This manual also gives some insight into the Pascal 68000 System structure, its components and its behaviour.

This manual is designed for programmers who have a working knowledge of Pascal. Detailed knowledge of 68000 UNIX is helpful but not essential.

Pascal was designed by Professor N. Wirth as a language for teaching structured programming techniques and as such is used widely in educational institutions. It has also gained popularity as a general-purpose language because it contains a set of language features that make it suitable for many different programming applications. Pascal has furthermore strongly influenced the development of several languages (e.g. ADA). The Pascal language includes a variety of control statements, data types, and predefined procedures and functions.

☛ Throughout this document the following notation is used:

Keywords and **predefined** identifiers are printed in bold face.

Syntactic variables as well as *UNIX components* are printed in italic font.

Metasymbols {,} and [,] are used for optional parts (0..∞ and 0..1), (,) bracket syntactical units and / separates alternatives. Terminal symbols are printed in roman face; to distinguish metasymbols from terminal symbols apostrophes ' are used if necessary.

2. Pascal Language

2.1. Supported Language

Pascal 68000 is an extended implementation of the Pascal language [1]. Specifically Pascal 68000 adheres to the Pascal language as described in the suggested ISO Standard [2]. This "draft Standard" is a cleaned up version of the original Pascal and has been submitted to ISO for acceptance.

Pascal 68000 was tested against a "Pascal Processor Validation Suite" ([3] ,[4]). These tests should give a good overview of the quality of the implemented system.

2.2. Deviations and Extensions

2.2.1. Deviations

Pascal 68000 deviates from the standard proposal in the following ways:

(1) Only the first 16 characters of an identifier are significant. The truncation of an identifier to 16 characters alters the meaning of a conforming program.

(2) Standard procedures and functions are not allowed as parameters, as in previous standard proposals. Identical results with minor loss in performance can be obtained by declaring user procedures. For example:

```
function userodd(i: integer) : boolean;
    begin
    userodd := odd(i)
    end;
```

(3) Type **real** is not yet implemented.

(4) Procedures that are to be used as parameters must be declared at main level.

(5) **dispose** is not yet implemented.

2.2.2. Extensions

2.2.2.1. Separate compilation

Pascal 68000 is able to compile so called **modules**, a collection of declarations, procedures and functions. The result of the compilation (an a.out object module) can be handled by the usual UNIX components, i.e. they can be stored in libraries, bound(loaded) with other a.out modules, etc.

The separate compilation feature is further supported by enabling import and export of variables, procedures and functions. Modules implemented in Pascal, C or assembler can be linked to Pascal 68000

modules. Procedures and functions are imported by using a directive in the heading: **extern** for Pascal- and assembler- and **externc** for C-procedures; they are exported implicitely by being defined on the outermost level within a module. The user is responsible for parameter and result compatibility.

Variables are imported or exported by using the newly introduced keywords **import** or **export** instead of **var**. The two predefined variables **input** and **output** are (per default) exported from a mainprogram, if they are listed in the program heading. If used, they have to be imported in a module. It is not possible to import or export labels!

The new syntax for a compilation unit is:

compilation_unit =
program *name* '(' *files* ')' ; *block* .
module *name* ; {*declaration*} .

block =
{*declaration*} *compound_statement*

2.2.2.2. Additional standard procedures

The following additional standard procedures are available:

addr
This function returns the address of the parameter, which is compatible with all pointer types.

convert
This function changes the type of the first parameter.

release, mark
Easier to implement than dispose, mark and release allow to use the heap as a stack. mark(p) stores the current value of the heap pointer in p. release(p) restores the heap pointer to p.

2.2.2.3. Underscore as letter

The character '_' is significant and can be used in forming identifiers.

2.2.2.4. Alternate symbols

There are two representations for comment symbols ('(*' , '*)' and '{' , '}') and for bracket symbols ('(.' , '.)' and '[' , ']').

2.2.2.5. Default case

In a case statement a default case can be defined. The both keywords **otherwise** and **else**: will be accepted.

2.2.2.6. Exponent

A lower case **e** may be used to represent real numbers.

2.2.2.7. Declaration

The order of declaration for labels, constants, types, variables, functions and procedures has been relaxed. Any order and any number of times declaration sections may be used. Furthermore, **import** and **export** variable declaration are implemented to support the separate compilation feature.

An identifier must be declared before it is used. Two exceptions exist to this rule:

(1) Pointer types may be forward referenced as long as the declaration occurs within the same type-definition-part

(2) Functions and procedures may be predeclared with a forward declaration.

The new syntax for a block is:

block = {*declaration*} *compound_statement*

declaration =
import (*names* : *type*) ; ... ;
/ **export** (*names* : *type*) ; ... ;
/ **var** (*names* : *type*) ; ... ;
/ **label** *label* , ... ;
/ **const** (*name* '=' *constant*) ; ... ;
/ **type** (*name* '=' *type*) ; ... ;
/ *function_declaration*
/ *procedure_declaration*

2.2.2.8. Hexadecimal constants

Hexadecimal integers are indicated by a preceding "#". The syntax for a hexadecimal integer is:

unsigned_number = *digit* {*digit*} / # *hexadigit* {*hexadigit*}
digit = 0/1/2/3/4/5/6/7/8/9
hexadigit = *digit* /A/B/C/D/E/F/a/b/c/d/e/f

2.2.2.9. Generic pointers

Generic pointers provide a tool for generalized pointer handling. Variables of type **address** can be used in the same manner as any other pointer variable with the following exceptions:

- generic pointers cannot be deferenced since there is no type associated with them.
- generic pointers cannot be used as an argument to **new**.
- any pointer can be assigned to a generic pointer. Use **convert** for assigning a generic pointer to a typed pointer.

2.2.2.10. Shorts

Type **short** is implemented. Subranges which fit in the range -2^{15} .. 2^{15-1} are treated as **shorts**.

2.2.2.11. Character constants

Certain non-printable characters may be represented according to the following table of escape sequences:

```
\\   backslash
\b   backspace
\f   form feed
\n   newline
\r   carriage return
\t   horizontal tabulator
\ddd
```

The escape sequence *ddd* consists of a backslash followed by 1,2, or 3 octal digits which are taken to specify the value of the desired character. If the character following the backslash is not one of those specified, the backslash is ignored.

2.3. Results of Validation Test

Pascal 68000 was tested against a "Pascal Processor Validation Suite". This suite consists of many programs which are classified as:

Conformance
: These programs adhere strictly to the ISO Standard. They should correctly compile and execute.

Deviance
: These programs are not allowed by the ISO Standard. The deviances (sometimes extensions) should be detected at compile or run-time.

Implementationdefined
: These programs contain implementation dependent features which are the responsibility of the implementors.

Errorhandling
: These programs contain errors which must be reported at compile or run-time.

Quality
: These programs attempt to assess quality. They should find out the limits of the system, the accuracy of the arithmetic, the ability of compiler diagnostics, etc.

The results of the tests are summarized in the next paragraphs. They should give a good overview of the quality of the implemented Pascal 68000 system.

2.3.1. Conformance Tests

ISO 6.1.5-2, 6.4.2.2-1, 6.4.6-1, 6.4.6-2, 6.4.6-3, 6.6.1-1, 6.6.6.2-1, 6.6.6.2-2, 6.6.6.2-3, 6.6.6.3-1, 6.7.1-2, 6.9.2-3, 6.9.4-4: **reals** are not yet implemented.

ISO 6.4.3.5-3: End of line marker is not inserted at the end of the line, if not explicitly done in the program.

ISO 6.5.1-1: **files** are not allowed in structured data.

ISO 6.6.3.4-2: Inconsistency in the implementation of procedure parameters.

ISO 6.6.5.3-2: **dispose** is not implemented.

ISO 6.7.2.2-5: Big constants can cause compiler crash.

ISO 6.8.3.9-1: Assignment to **for** control variable is done after evaluation of initial expression (i.e. before evaluation of the terminating expression).

2.3.2. Deviance Test

ISO 6.1.2-1: Reserved word **nil** may be redefined.

ISO 6.1.5-6: A lower case **e** may be used in real numbers.

ISO 6.1.7-11: A null string is accepted by the compiler.

ISO 6.2.2-4, 6.3-6, 6.4.1-3: Scope error not detected by the compiler.

ISO 6.6.2-5: A function without an assignment to the function value variable in its block compiles and runs.

ISO 6.8.2.4-2, 6.8.2.4-3, 6.8.2.4-4: A **goto** between branches of a statement is permitted.

ISO 6.8.3.5-12: Extension: subranges allowed as **case** constant element.

ISO 6.8.3.9-2, 6.8.3.9-3, 6.8.3.9-4, 6.8.3.9-16: An assignment can be made to a **for** statement control variable and in fact changes the value, but has no effect on the number of repetitions.

2.3.3. Implementationdefined

ISO 6.4.2.2-7: **maxint** is 2 147 483 647.

ISO 6.4.3.4-4: **set** bounds are 0 and 255.

ISO 6.6.6.1-1: Standard procedures and functions not allowed as formal parameters.

ISO 6.6.6.2-11, 6.9.4-5, 6.9.4-11, 6.11-2: **reals** not yet implemented.

ISO 6.8.2.2-1, 6.8.2.2-2: A variable is selected before the expression is evaluated in an assignment statement.

ISO 6.9.4-11: Default field width specification: 10 for **integers** and 5 for **booleans**.

ISO 6.10-2: Rewrite on the standard output file is permissible.

ISO 6.11-1, 6.11-3: Alternative comment delimiters and bracket delimiters have been implemented. No other alternative symbols have been implemented.

2.3.4. Error Handling

ISO 6.2.1-7, 6.6.2-6, 6.8.3.9-5, 6.8.3.9-6: Uninitialized or undefined variables are not detected.

ISO 6.4.3.3-5, 6.4.3.3-6, 6.4.3.3-7, 6.4.3.3-8: No runtime checks are performed on the tag field of variant records.

ISO 6.4.6-7, 6.4.6-8, 6.7.2.4-1: No bounds checking is performed on overlapping set operands.

ISO 6.6.2-6: The use of a function without an assignment to the function value variable is permitted.

ISO 6.6.5.2-6, 6.6.5.2-7: Fail because I/O has not been implemented strictly according to the standard.

ISO 6.6.5.3-3, 6.6.5.3-4, 6.6.5.3-5, 6.6.5.3-6: **dispose** is not implemented.

ISO 6.6.5.3-7, 6.6.5.3-8, 6.6.5.3-9: No checks are inserted to check pointers after they have been assigned a value using the variant form of new.

ISO 6.6.6.4-7: No bound checks are inserted for the **succ**, **pred** or **chr** functions.

ISO 6.7.2.2-6: Big constants cause compiler crash.

ISO 6.8.3.9-5, 6.8.3.9-6: The **for** control variable is not invalid after the execution of the **for** statement.

ISO 6.8.3.9-17: Two nested **for** statements with the same control variable are permitted, but do not run into an infinite loop.

2.3.5. Quality Measurement

ISO 5.2.2-1: Identifiers are not distinguished over their whole length.

ISO 6.1.3-3: Only the first 16 characters of an identifier are significant.

ISO 6.1.8-4: Unclosed comment not detected.

ISO 6.2.1-8, 6.2.1-9, 6.5.1-2: Large lists of declarations can be made in each block.

ISO 6.4.3.2-4: An array with **integer** index type is not permitted

ISO 6.4.3.3-9: The variant fields of a **record** occupy the same space, using "exact correlation".

ISO 6.4.3.4-5: Warshall's algorithm runs.

ISO 6.6.1-7: Procedures can be nested to a depth exceeding 15.

ISO 6.7.2.2-4: **div** and **mod** have been implemented consistently. **mod** returns remainder of **div**.

ISO 6.8.3.5-2: The **case** constants must be compatible with the **case** index, but do not have to be of the same type if the **case** index is a subrange.

ISO 6.8.3.5-8: Large **case** statement is permitted.

ISO 6.8.3.9-18: A range check is performed in a **case** statement after a **for** statement, to check the value of the **for** control variable.

ISO 6.8.3.9-20, 6.8.3.10-7: **for** and **with** statements can be nested to a depth exceeding 15.

ISO 6.9.4-10: The output buffer is flushed.

ISO 6.9.4-14: Recursive I/O is permitted, using the same file.

3. Pascal 68000 under UNIX

3.1. Creating and executing a program

The usual way to create and execute a program is realized by entering the following commands to the 68000 UNIX operating system. With each command, you include information that further defines what you want the system to do. Of prime importance is the file specification, which indicates the file to be processed. You can also specify qualifiers that modify the processing performed by the system ($ is the system prompting symbol).

A program is entered or corrected by any editor of the user's taste. The file name of Pascal source programs must have the suffix '.p'.

```
$ edit <name>.p CR
```

The *pc* [5] command compiles and links the Pascal program. The resulting object module is left in <name>.

```
$ pc -o <name> <name>.p CR
```

The program is loaded and run by:

```
$ name CR
```

The only program parameters supported by the Pascal language are files. There are three ways to associate an (external) UNIX file specification with an (internal) Pascal file specification. The standard Pascal files **input** and **output** are always associated with the logical UNIX files *stdin* and *stdout*. Their comfortable and flexible use is described in [6]. All other Pascal files can be associated with any UNIX file either by assignment within a commandline:

```
$ name pascalfile_1=UNIX_file_specification \
       pascalfile_2=UNIX_file_specification [1] \
       . . . CR
```

or - if an assignment is missing - interactively:

```
$ name pascalfile_2=UNIX_file_specification CR
pascalfile_1 ? UNIX_file_specification CR
 . . .
```

Example:

```
$ ed example.p CR
$ pc -o example example.p CR
$ example eingabe=/dev/tty CR
ausgabe ? example.aus CR
```

[1] As these assignments are considered as one argument each, there must be no blanks before or after the '=' sign.

It is advisable to define a shell procedure for a more convenient file assignment especially if some of the files are "fixed" or work files.

3.2. Support of UNIX Facilities

The user has full access to all UNIX system calls as well as files. The system calls can be accessed just like C procedures (see below). The objects are to be found in the standard library. The standard procedure **halt** provides a means to return an "exit code" to the system. Furthermore the "system variables" _argc and _argv are provided for access to the command arguments. _argc indicates the number of arguments whereas _argv references an array of argument strings. The variables can be declared as

```
{$t- }
type
   argva = array[1..n] of ^ string;    { n >= _argc }

import
   _argv : ^ argva;
   _argc : integer;
```

where string is a null-terminated character array (C convention). Remember that the file specifications are command arguments too, i.e. _argv[1]^[1] is the first character of the program name. If you access _argv^, it is essential to switch off the pointertest.

The C preprocessor *cpp* [7] may be used without limitations.

As stated earlier C and assembler modules can be loaded with Pascal. When connecting Pascal modules with C modules you should take care of parameter compatibility (see 8.) and of identifiers since C prefixes the identifiers of export variables and procedures with a "_" (i.e. a variable identified in Pascal as _argc would be called argc in C). Moreover, you must be sure that the C modules do not use the *sbrk/brk* system call which interferes with the Pascal heap.

3.3. Limitations of Pascal 68000

- Because of the separate compilation feature, a missing **reset** or **rewrite** cannot be detected by the compiler and will most probably cause the program to crash with an address error at the first attempt to access the corresponding file-variable.
- Currently, a **reset** on **input** or a **rewrite** on **output** have no effect at all. Therefor, it is not possible to reposition **input** or **output** to the beginnig of the associated file.
- Actually, code produced per procedure must not exceed 10 KB. This limit is a compiler constant and can be changed by recompiling pass2.
- With the current 68000 processor it is not possible to have a dynamically growing stack (see "Unterschiede vun MUNIX to UNIX V7; M. Uhlenberg"). Programs, that crash with stackoverflow, can be

restarted after having enlarged the stacksize by means of the command *stksiz*.

- The datasize of a program (heap and stack) is limited to a total of ca. 240 KB (see "Unterschiede ... ").

4. Compiler options

Compiler options are given by using "{$...}". Each option consists of a lower or upper case letter followed by "+" or "-". Options are separated by commas. There must be no blanks between "{" and "$" or between a comma and the succeeding option. The following options are supported:

A/a +/-	a+ produces an assembler listing
D/d +/-	d+ produces code for pointer, subrange check and line numbers
E/e +/-	e+ will suppress extension warnings
P/p +/-	p+ code for profiling is generated (see *prof* [8])
T/t +/-	t+ produces code for pointer check
W/w +/-	w+ will suppress warning messages

Defaults: {$a-,d-,e-,p-,t-,w-}

Options appearing before the **program** or **module** symbol can be overwritten by options given in the *pc* command.

5. Error handling

Errors are detected and reported by several components of the Pascal 68000 system:

- preprocessor *cpp*, see *pc*
- compiler
- loader *ld*, see *pc*
- run-time system

5.1. Compiletime Detection of Source Errors

pass1 detects syntactical and some semantical errors and (optionally) deviations from the standard language definition. Errors that are not detected are listed in 2.3.4. *pass1* does not produce error messages or a listing but compiles a condensed version of the diagnostics that will be printed in a readable form by an extra pass *perror*. If only warnings are issued compilation proceeds otherwise it will be terminated.

The option "-L" (see *pc*) produces a full listing with error messages. The default is a listing containing the offending line, its predecessor and the readable (and hopefully understandable) error messages. Sometimes an error causes several messages to be printed in which case all but the first one can be ignored.

pass2 detects no source errors, but might report a compiler error (though of course it should not). Please let us know if you get an compiler error or an unknown error message.

5.2. Other Errors Detected at Compiletime

During compilation, UNIX resources can be exhausted, e.g. file system, process table, or memory overflow, etc. Furthermore UNIX can deny access to files. Extensive treatment of these errors is beyond the scope of this manual. They are described in the UNIX documentation.

5.3. Runtime errors

Errors occurring at run time are always fatal, i.e the program will report an error message, dump the core file and then abort. With the help of the UNIX debugger *adb* [9] the user can generate a (partially) symbolic post mortem dump which indicates the location of the fatal error, the number of the corresponding source line (if the debug option was on), the dynamic calling sequence, etc.

The error message should comprise a sufficient diagnosis of the error detected. As far as file access is concerned, the errors are mostly reported by the UNIX system and must be investigated using the documentation. Other messages indicate programming errors such as divide by zero, integer overflow , etc.

6. Pascal System Components

Figure 6.1 gives an overview of the Pascal system components. The preprocessor is the same as the C-compiler's, and does macro preprocessing and including of source files.

The first pass *pass1* does the lexical, the syntactical and the semantical analysis. It constructs a parse tree for each block and outputs it.

There is an extra pass *perror* to produce a source listing if requsted, and to print error messages based on the diagnostics compiled by *pass1*.

The second pass *pass2* does the code generation. The output of *pass1* is read in and an identical parse tree as in *pass1* is built up. The generated code is output in several files.

The third pass *c2* is the same as the C-Compiler's and collects the code distributed on various files. It produces one file containing the object code in *a.out* format [10]".

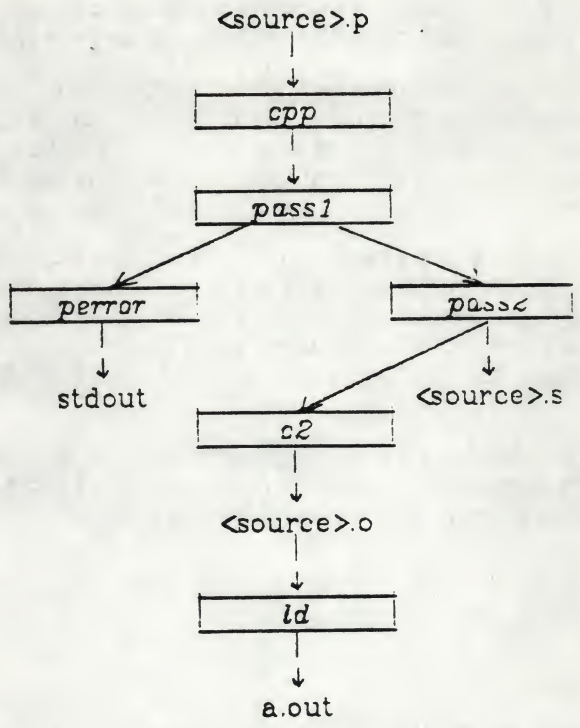


Figure 6.1: Pascal System Components

6.1. Hardware and Software Environment

Pascal 68000 runs on two quite similar 68000 UNIX systems, PERKEO [11] and QU68000 [12].

Features of these systems are:

- actually supported microprocessor Motorola 68000
- large memory (≥ 0.5 MB)
- hard disk (≥ 10MB)
- memory management unit

UNIX [13] [14] is a time-sharing system and provides all such features needed by Pascal 68000. Of great importance are the file system, the command interpreter (shell), and the object module management (*ar*, *ld* [15]). But of equal, if not greater, weight are all those UNIX components which make life easier: *ed*, *re*, *ped*, *make*, etc.

6.2. Pass1

The first pass does lexical analysis, parsing, declaration handling, tree building, and some optimization. This pass is largely machine independent.

The lexical analysis is a conceptually simple procedure that reads the input and returns the tokens of the Pascal language as it encounters them: identifiers, constants, operators and keywords. Comments must be skipped. Decimal and hexadecimal constants, characters and strings must be properly dealed with.

The first pass parses, as the original Pascal-P4 compiler, the tokens in a top down, left right, recursive descendent fashion. During the processing of a declaration part a symbol table is built up, addresses will be allocated to variables and procedures, and the semantic of the declaration is checked. Besides this, information is prepared for the debugger *adb* (or *pbug* [16]).

During the processing of the statement part a parse tree is built. The proper use of operators and operands is checked. Some complex syntactical structures are broken down into simpler ones.

6.3. Pass2

The second pass generates 68000 machine code and related information from the source program represented by the parse tree; these will be combined to a *a.out* object. In addition it completes the debugger information prepared by *pass1*.

Code generation is done in two steps. The first one traverses the parse tree generating pseudo instructions for a hypothetical stack machine. This step is rather simple, machine independent and straightforward. The

second step deals with machine specific tasks such as address computation, register allocation and 68000 code generation, thus interpreting the pseudo instructions in a real existing environment. Step one and two take place in parallel for efficiency reasons; the generation of a pseudo instruction is replaced by calling a procedure implementing this instruction.

pass2 yields different kinds of output. Three files contain binary information ready to be combined to an *a.out* object by a third pass. A fourth file may contain the generated machine code in an assembler-like representation. On another file debugger information is prepared. And there might be diagnostic messages indicating a compiler error.

6.4. Cross Reference

The cross reference program *xref6* produces first an overview of your Pascal source program in form of a static nesting procedure diagram, and second the usual cross-reference list with identifiers and line numbers.

xref6 is implemented as an independent component. There are several reasons for doing so:

- The compiler is not bothered with cross referencing.
- The multiprogramming (parallel processing) of the system can be exploited.
- Many programming errors can be found with the help of a cross-reference listing; i.e. it is not necessary to start the big, resource consuming, compiler.

6.5. Prettyprinter

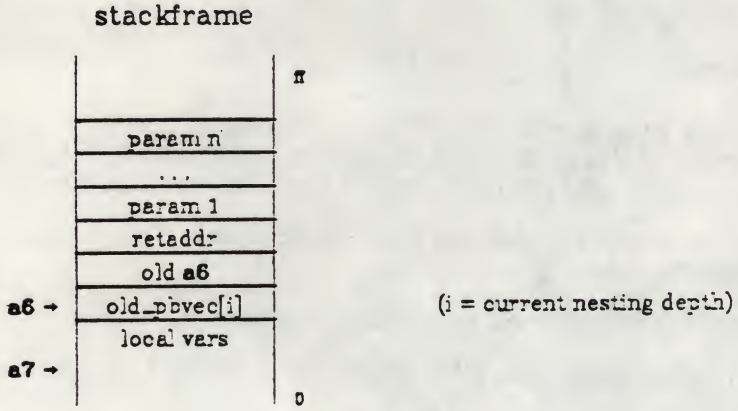
One important aspect about Pascal coding style is consistency, although styles certainly differ from one programmer to the next. *pretty* prints a Pascal program in a "pretty" form with standard indentations, see the prettyprinting example.

pretty provides several options because no prettyprinting style can please everyone; by allowing complete control over the process, one can achieve pleasing results. Unfortunately, comments are not too well supported. But *pretty* can "beautify" your program source both after your first typing and after corrections.

For more details see [17].

7. Calling Conventions

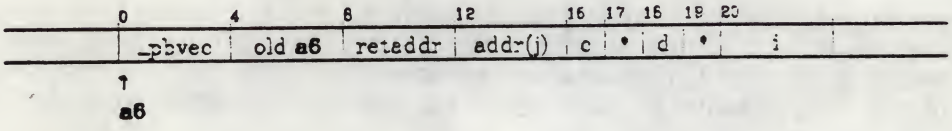
Procedure calls are realised by a commonly used mechanism defining stackframes, which are allocated or deallocated as a procedure is activated or deactivated, respectively. In our implementation four registers and a vector of pseudo registers are used: **a7** is used as stackpointer, **a6** and **a5** reference the current and the global stackframe base, respectively. **d0** returns a function result and the system variable _pbvec[0..maxdepth] stores the base addresses of all currently accessible stackframes. The layout of a stackframe is shown below.



A parameter is passed by-reference or by-value depending on whether it was declared as **var** parameter or not. The representation of parameters in memory is the same as for other variables with the exception that they are always word aligned, i.e. a parameter occupying an odd number of bytes in memory will always be followed by a byte of undefined storage.

Example:

stackframe for
procedure p(**var** j:**integer**; c,d:**char**; i:**integer**);
after procedure entry

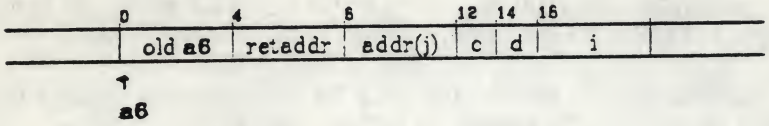


• undefined

The C interface provided as an extension differs from the Pascal parameter passing conventions only in the treatment of one-byte values; following the C semantics, data occupying one byte of storage are extended to (unsigned) word values and then treated like a **short**.

Example:

stackframe for
procedure p(**var** j:**integer**; c,d:**char**; i:**integer**); externc;
after procedure entry



where c & d are pushed as zero padded **short** items
(at our installation Pascal **short** corresponds with C **int**)

8. Data Representation and Allocation

In a.out modules program data fall into three segments: the text segment, the data segment and the bss segment. Pascal 68000 uses only two of them: the text segment contains program code and constants whereas static data (**exported**ed by a module) are stored in the bss segment. Automatic and dynamic data are allocated at runtime in stack and 'free memory' managed by the *brk/sbrk* system calls, respectively.

To cope with alignment, one general rule can be stated: any data allocated more than one byte is aligned on a word (2 bytes) boundary.

Variables of scalar and pointer types are allocated storage space as summarized in table 8.1. Variables of subrange types are allocated as variables of the associated scalar types. For example, a type 1..10 is considered a subrange of **short** and therefor allocated a word. The structured types are stored as described below. The attribute **packed** is ignored.

Type	Storage	Allocation
character	8 bits (1 byte)	Byte
boolean	8 bits (1 byte)	Byte
short	16 bits (1 word)	Word
integer	32 bits (1 longword)	Word
real		
enumerated	8 bits (1 byte) if type contains 256 elements or less; 16 bits (1 word) otherwise	Byte if type contains 256 elements or less; Word otherwise
pointer	32 bits (1 longword)	Word

Table 8.1: Storage of Scalar and Pointer Types

A set is allocated storage depending on the ordinal value of its largest element: the number of bytes it occupies is equal to the ordinal value rounded up to the nearest byte boundary. Since the size of a set is limited to 256 elements, with ordinal values from 0 to 255, a set occupies at most 32 bytes.

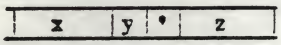
An array is stored and aligned according to the type of its components. For example, each element of a character array is stored in a byte and aligned on a byte boundary; if the array has more than one component then the total array is aligned a on word boundary. Similarly, each element of an array of set of 3..21 occupies three bytes and is aligned on a word boundary.

Constant strings are internally terminated by a binary 0; i.e. they adhere to the C convention.

Records are stored and aligned field by field according to to the type of the field. F'or example, a variable of type

```
record
  x: integer;
  y: boolean;
  z: integer
  end;
```

is aligned on a word boundary because it occupies more than one byte of storage. The figure beneath shows how this variable is stored:



• not used

9. Appendix

9.1. Examples

9.1.1. Sample program

```
 1 program ackermann(input,output);
 2  var x, y: integer;
 3  function ack(i,j: integer): integer;
 4   begin
 5   if       i = 0 then  ack:=j+1
 6   else  if j = 0 then  ack:=ack(i-1,1)
 7         else      ack:=ack(i-1,ack(i,j-1))
 8   end;
 9
10  begin
11   repeat
12    writeln(output,'Enter two integers. Terminate with zero.');
13    read(input,x,y);
14    writeln(output,'acker(',x:0,',',y:0,') = ',ack(x,y):0)
15    until x=0
16  end.
```

9.1.2. Crossreference

```
*** cross-reference-list (zfe fi sar 121  may 80) ***


*** static structure of procedures / functions ***

  1-  16 I ackermann
  3-   8 I . ack
 10-  16 I *code*



f = formal parameter   k = constant      m = main program
r = record field       t = type          v = variable
p = procedure / function

ack                    3p    5    6    6    7    7    7   14
ackermann              1
i                      3f    5    6    7    7
input                  1    13
integer                2    3    3
j                      3f    5    6    7
output                 1    12   14
read                  13
x                      2v   13   14   14   15
y                      2v   13   14   14
***end of cross-reference
```

9.1.3. Prettyprinting

The prettyprinted sample program using default options is:

```
program ackermann(input, output);

var
        x,
        y: integer;


function ack(i, j: integer): integer;
  begin
  if i = 0 then ack:=j + 1
  else
    if j = 0 then ack:=ack(i - 1, 1)
    else ack:=ack(i - 1, ack(i, j - 1))
  end (*ack*);

begin (*ackermann*)
repeat
  writeln(output, 'Enter two integers. Terminate with zero.');
  read(input, x, y);
  writeln(output, 'acker(', x: 0, ',', y: 0, ') = ', ack(x, y): 0)
until x = 0
end (*ackermann*).
```

9.1.4. Separate Compilation

```
 1 program ackermann(input,output);
 2 var x, y: integer;
 3
 4 import counter: integer;
 5
 6 function ack(i,j: integer): integer;  extern;
 7
 8 begin
 9  repeat
10    writeln(output,'Enter two integers. Terminate with zero.');
11    read(input,x,y);
12    counter:=0;
13    writeln(output,'acker(',x:0,',',y:0,') = ',ack(x,y):0);
14    writeln(output,'number of calls=',counter:5);
15  until x=0
16 end.
```

```
 1 module ack;
 2
 3 export counter: integer;
 4
 5 function ack(i,j: integer): integer;
 6  begin
 7  counter:=counter+1;
 8  if       i = 0 then  ack:=j+1
 9  else if  j = 0 then  ack:=ack(i-1,1)
10       else      ack:=ack(i-1,ack(i,j-1))
11  end;
12 .
```

9.2. Size of Components

Component	Language	Locs	Code Size (bytes)
pass1	Pascal	3800	64000
pass2	Pascal	3000	46000
xref6	Pascal	500	15000
pretty	Pascal	1100	28000

9.3. Standard Procedures and Functions

For a detailed description see [2].

Procedure	*Parameter*	*Result*	*Function*
abs(x)	**integer** **real** not yet	same as x	Computes the absolute value of x.
addr	any type	address	Type address is compatible with all pointer types.
arctan(x)	**real**	**real**	not yet
chr(x)	**integer**	**char**	Returns the character whose ordinal number is x.
convert(a,t)	any type	t	Returns value of a with type t.
cos(x)	**real**	**real**	not yet
eof(f)	**file**	**boolean**	End of file encountered. **true** only when the file position is after the last element in the file.
eoln(f)	**file**	**boolean**	End of line encountered. **true** only when the file position is after the last character in a line. The value of f^ is a space.
exp(x)	**real**	**real**	not yet
get(f)	**file**		Moves the current file position to the next element.
halt(x)	**integer**		Terminates the execution of the program and returns the value of x. See also the system call **exit**(2).
halt			same as **halt**(0).
ln(x)	**real**	**real**	not yet
mark(x)	**pointer**		Stores the current value of the heap pointer into x.

Procedure	*Parameter*	*Result*	*Function*
new(p)	**pointer**		Allocates heap memory and returns the address in p.
new(p,t1,..tn)			Allocates heap memory to pointer. p^ must be a record with variants.
odd(x)	**integer**	**boolean**	Returns **true** if the integer x is odd; **false** otherwise.
ord(x)	any scalar type except **real**	**integer**	Returns the ordinal (integer) value corresponding to the value of x.
pack(a,i,z)			z:=a[i..n]; where i..n: index range of z.
page(f)	**file**		skip to the top of a new page before printing the next line of the textfile f. The default for f is **output**.
pred(x)	any scalar type except **real**	same as x	Returns the predecessor value of x.
put(f)	**file**		Appends f^ to the file f. The default for f is **output**.
read(f,x)	**file** type of x depends on the filetype		Reads the value of x from the file f. The default for f is **input**.
readln(f)	**file**		Skips to the beginning of the next line. The default for f is **input**.
read(f,x1,..,xn)			same as **read**(f,x1); **read**(f,x2,..xn)
readln(f,x1,..,xn)			same as **read**(f,x1,..,xn); **readln**(f)

Procedure	*Parameter*	*Result*	*Function*
release(x)	**pointer**		Loads the heap pointer with the value of x.
reset(f)	**file**		Resets file for reading.
round(x)	**real**	**real**	not yet
sin(x)	**real**	**real**	not yet
sqr(x)	**integer** **real** not yet	same as x	Computes the square of x.
sqrt(x)	**real**	**real**	not yet
succ(x)	any scalar type except **real**	same as x	Returns the successor value of x.
trunc(x)	**real**	**integer**	not yet
unpack(z,a,i)			a[i..n]:=z; where i..n: index range of z.
write(f,x)	**file** type of x depends on the filetype		Writes the value of x to the file f. The default for f is **output**.
writeln(f)	**file**		Starts a new line. The default for f is **output**.
write(f,x1,..,xn)			same as **write**(f,x1); **write**(f,x2,..xn)
writeln(f,x1,..,xn)			same as **write**(f,x1,..,xn); **writeln**(f)

9.4. Syntax Equations

```
letter =    a/b/c/d/e/f/g/h/i/j/k/l/m/n/o/p/q/r/s/t/u/v/w/x/y/z
            /A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T
            /U/V/W/X/Y/Z/_

digit =     0/1/2/3/4/5/6/7/8/9

name =              letter { letter / digit }
constant_name =     name
type_name =         name
variable_name =     name
  ...
names =             name , ...
label =             unsigned_number

compilation_unit =
    program '(' names ')' ; block .
  / module name ; {declaration} .

block =     {declaration} compound_statement

declaration =
    / import ( names : type ) ; ... ;
    / export ( names : type ) ; ... ;
    / var    ( names : type ) ; ... ;
    / label  label , ... ;
    / const  ( name '=' constant ) ; ... ;
    / type   ( name '=' type ) ; ... ;
    / function_declaration
    / procedure_declaration

constant =
    [ sign ] ( unsigned_number / constant_name )
  / character_string

sign = + / -

unsigned_constant =
      unsigned_number / character_string
    / constant_name / nil

unsigned_number =
    digit {digit}  / # hexadigit {hexadigit}

hexadigit =             digit /A/B/C/D/E/F/a/b/c/d/e/f
character_string =      /*characters enclose by ' '*/
```

---------- t y p e ----------

```
type =              type_name / new_type
new_type =          simple_type / structured_type / ^ type_name

simple_type =       ordinal_type / real [2]
```

```
ordinal_type =
      '(' names ')'
    / constant .. constant
    / ordinal_type_name

structured_type =
      [ packed ] unpacked_structured_type
    / structured_type_name

unpacked_structured_type =
      array '[' ordinal_type , ... ']' of type
    / file of type
    / record [ field_list [ ; ] ] end
    / set of ordinal_type

field_list =
      fixed_part [ ; variant_part ]
    / variant_part

fixed_part =      ( names : type ) ; ...
variant_part =    case [ tag_field_name : ] tag_type of variant ; ...
variant =         case_constant_list : '(' [ field_list [ ; ] ] ')'

case_constant_list =    case_constant , ...
case_constant =         constant [ .. constant ]
```

---------- expression ----------

```
variable = ( variable_name / field_name )
    { '[' expression , ... ']'
    / . field_name
    / ^ }

factor =
      variable
    / unsigned_constant
    / function_name [ '(' actual_parameter , ... ')' ]
    / set
    / '(' expression ')'
    / not factor

actual_parameter =
    expression
    / procedure_name / function_name

set    =    '[' [ member , ... ] ']'
member = expression [ .. expression ]

term =    factor
    { ( * / '/' / div / mod / and ) factor }

simple_expression = [ sign ] term
    { ( + / - / or ) term }
```

---

[2] Type **real** is not yet implemented.

```
expression = simple_expression
    [ ( <> / = / < / > / <= / <= ) simple_expression ]
```

---------- procedure ----------

```
procedure_declaration =
    procedure_heading ;
    ( block / directive )

function_declaration =
    function_heading ;
    ( block / directive )

directive =                 forward / extern / externc
procedure_heading =         procedure name [ '(' parameter ; ... ')' ]

function_heading =          function name ['(' parameter ; ... ')']
                    :result_type

parameter =
     function_heading / procedure_heading
    / names : type
    / var names : ( type_name / array_type )

array_type [3] =
    array '[' index_type ; ... ']' of (type_name / array_type )

index_type =
    name .. name : ordinal_type_name
```

---------- statement ----------

```
statement = [ label : ]
     compound_statement
    / case expression of case_element ; ... [ ; ] end
    / for name := expression
          ( to / downto ) expression
          do statement
    / goto label
    / if expression then statement
          [ else statement ]
    / repeat statement ; ... until expression
    / while expression do statement
    / with variable , ... do statement
    / ( variable / function_name ) := expression
    / procedure_name [ '(' actual_parameter , ... ')' ]

compound_statement =
    begin statement ; ... end
```

[3] array_type (i.e. conformant arrays) are not yet implemented.

case_element =
    ((*case_constant* / **else**) : / **otherwise**)
    statement

9.5. Description of the Parse Tree

A Pascal source program is translated into a sequence of parse trees representing the blocks of the program. A parse tree is like a pascal block a list of statements, which in turn are build by lists of expressions and statements. The following basic data structures reflect the formal structure of a tree; the description is enhanced by some informal information given below.

Data structures describing the tree:

```
type

 listclass =     (lexpr, lstatmt);
 statmtclass = (sassign,sbegin,scall,scase,sdownto,sgoto,
                sif,slabel,srepeat,sto,swhile,swith);
 exprclass =   (
    opstore, opdotdot, opset, opnop,
    oprange, oplength, opcomma, opchkp, opchks, opchkr,
    opcase1, opaddr, opaddrof, oplcst, oplind, oplptr,
    opperiod,opindex, opindex1, opaddrcst,
    opcall, opcallc, opcallf, opproc,
    opcmpa, opcmpb, opcmpc, opcmpf, opcmpi, opcmps, ,
    opcmpm, opcmpp, opin,
    opabsi, opabsf, opabss, opoddi, opodds,
    opsqri, opsqrf, opsqrs,
    opinci, opincs, opincc,
    opdeci, opdecs, opdecc,
    opaddf, opsubf, opmulf, opdivf
    opaddi, opsubi, opmuli, opdivi, opmodi,
    opadds, opsubs, opmuls, opdivs, opmods,
    opaddm, opsubm, opmulm, opand, opor,
    opnegi, opnegs, opnegf, opnot
    opctoi, opctos, opitoc  opitos, opstoi, opstoc,
    opftos, opftoi, opitof, opstof
    oplt, ople, opeq, opge, opgt, opne,
    optrue, opfalse, opltu, opleu, opgeu, opgtu );

 listp =         ^ listrec;
 statmtp =       ^ statmtrec;
 exprp =         ^ exprrec;

 exprrec =
    record
     case ekind : exprclass of
      oplcst   : (ll: integer);
      opaddr   : (l,r: integer);
      oplength : (link: exprp; rr: integer);
      opaddi   : (llink,rlink: exprp)
    end;

 listrec =
    record
     next: listp;
     case lkind: listclass of
```

```
        lstatmt: (statmt: statmtp);
        lexpr:  (expr: exprp)
    end;

statmtrec =
    record
     case skind: statmtclass of
      sbegin, sdownto, sgoto, sif, slabel,
      srepeat, sto, swhile, swith:
            (sline: integer; slist: listp);
      scall,sassign:
            (eline: integer; sexpr: exprp)
    end;
```

The expression nodes oplcst, opaddr, oplength and opaddi listed in exprclass are representatives of four subclasses of exprclass, the classification of which can be seen in the following pass1 fragment (*outprint.p*):

```
case ekind of
    oplcst:
            outint(ll);

    opaddr, opcase1, oplptr, oprange:
            begin outint(l); outint(r); end;

    opinci, opincs, opincc,
    opdeci, opdecs, opdecc,
    opaddrcst, oplength, opindex1, opperiod :
            begin outint(rr);
            outexpr (link); end;

    opcallf,
    opdotdot,opadds , opsubs , opdivs, opmuls,
    opaddf , opaddi , opaddm , opand , opcmpa,
    opcmpb , opcmpc , opcmpf , opcmpi, opcmpm,
    opcmpp , opcomma, opdivf , opdivi, opin,
    opindex, opmodi , opmulf , opmuli, opmulm,
    opor  , opstore, opsubf , opsubi, opcallc,
    opsubm , opcall , opchkr , opmods, opcmps:
            begin outexpr(llink); outexpr(rlink) end;

    opabsi, opabsf, opabss, opoddi,
    opset , opsqri, opsqrf, opsqrs, opodds,
    opproc, opchkp, opchks, opctoi, opeq  , opge  ,
    opgt  , opitoc, opitof, opitos, ople  , opctos,
    oplind, oplt  , opne  , opnegi, opnegf,
    opnegs, opnot , opftos, opstoc, opstoi,
    opstof, opftoi, opaddrof:
            outexpr (llink);
    otherwise write('*** error *** unknown expression')
              end (*case*)
```

Statements are translated into a list of expressions or statements.

sbegin	statement_list
sif	expression statement statement
swhile	expression statement
srepeat	statement_list expression
slabel	statement
sgoto	
sto,sdownto	expression expression statement
swith	expression_list statement
scall,	
sassign	expression

All the expression nodes are listed together with their successor(s) named by a self-describing mnemonic. The names of non terminal nodes are similar to those in the Pascal syntax definition. The types of the source operands turn up as implicit type information in the expression nodes: **integer** (i), **short** (s), **real** (f), **set** (m) and **char** (c): for example opaddi, opadds, etc. (Note: opcallc is "call extern C procedure" , opcallf if "call formal procedure").

1) variable

opaddr	level (l) , offset (r)
oplptr	level (l) , offset (r)
oplind	variable (llink)
opperiod	variable (link) , address_increment (rr)
opindex	variable (llink) , index (rlink)
opindex1	index_expression (link) , elementsize (rr)
oppproc	opaddr
opaddrof	variable
oplcst	constant(ll)

2) coercion

opctoi, opitoc, opitof, opitos, opctos,
opftos, opstoc, opstoi, opstof, opftoi
expression (llink)

3) data operation

opaddi, opsubi, opdivi, opmuli, opmodi,
opadds, opsubs, opdivs, opmuls, opmods,
opaddf, opsubf, opdivf, opmulf,
opaddm, opmulm, opsubm,
opand, opor
left_expression(llink), right_expression(rlink)

opnegi, opnegs, opnegf,
opnot expression (llink)
opinci, opincs, opincc,
opdeci, opdecs, opdecc
expression (link) , inc/decrement (rr)

4) data relations

```
opeq, opne, opge, opgt, ople, oplt,
opcmpa, opcmpb, opcmpc,
opcmpi, opcmps, opcmpf,
opcmpp, opcmpm
    left_expression(llink),right_expression(rlink)

opin  element_expression(llink),set_expression(rlink) [4]
```

5) procedure call

```
opcall, opcallf, opcallc
    proc_address(llink),parameter_list(rlink)
opcomma
    parameter_list(llink),parameter_expression(rlink)
```

6) miscellaneous

```
opdotdot
 from_elem_expression(llink),to_elem_expression(rlink)
opset            element_expression(llink)
oplength  expression (link) , representation (rr)
opcomma[5] call_list (llink) , procedure_call (rlink)
opchkp           variable
opchkr           expression (llink) , range (rlink)
oprange          lower (l) , upper (r)
```

[4] the set operations opin, opdotdot & opset are always preceded by oplength. Expressions representing set elememts must evaluate to a value in the range 0..255.

[5] read & write procedures with variable parameter number are realized by a call_list

9.6. PC - Pascal Compiler

NAME

pc – Pascal compiler

SYNOPSIS

pc [option] ... file ...

DESCRIPTION

pc is the UNIX Pascal compiler. It accepts several types of arguments:

Arguments whose names end with '.p' are taken to be Pascal source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.p'. The '.o' file is normally deleted, however, if a single Pascal program is compiled and loaded all at one go.

The following options are interpreted by *pc*. See *ld*(1) for load options.

–c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

–d Switch on the debug mode.

–e Suppress extension warning messages.

–n Suppress execution ("dry run"). *pc* commands.

–o *output*
Name the final output file *output*. If this option is used the file 'a.out' will be left undisturbed.

–p Arrange for the compiler to produce code which counts the number of times each routine is called, also, if loading takes place, replace the standard startup routine by one which automatically calls *monitor*(3) at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

–w Suppress warning messages.

–D*name=def*

–D*name*
Define the *name* to the preprocessor, as if by '#define'. If no definition is given, the name is defined as 1.

–I*dir* Files of '#include' type whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in –I options, then in directories on a standard list.

–L Addtionally generates listings on corresponding files suffixed '.l'.

–P Run only the macro preprocessor and place the result for each '.p' file in a corresponding '.i' file and has no '#' lines in it.

–S Compile the named Pascal programs and leave the assembler-language output on corresponding files suffixed '.s'.

–T Trace and print *pc* commands. Temporary files are not deleted.

–U*name*
Remove any initial definition of *name*.

Other arguments are taken to be either loader(ld) option arguments, or Pascal-compatible object programs, typically produced by an earlier *pc* run, or perhaps libraries of Pascal-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to

produce an executable program with name **a.out**.

FILES

file.i	preprocessor output file
file.l	error and listing file
file.o	object file
file.p	input file
file.s	assembler listing file
a.out	loaded (linked) output
/tmp/ptm?	temporaries for *pc*
/lib/cpp	preprocessor
/lib/pass[12]	compiler for *pc*
/lib/c2	compiler pass3
/lib/perror	prints errors and listing
/lib/prt0.o	runtime startoff
/lib/mprt0.o	startoff for profiling
/lib/libp.a	pascal runtime-support
/lib/libc.a	standard library, see *intro*(3)
/usr/include	standard directory for '#include' files

SEE ALSO

K. Jensen and N. Wirth *Pascal User Manual and Report* Springer Verlag 1978
monitor(3), prof(1), adb(1), ld(1)

DIAGNOSTICS

The diagnostics produced by *pc* itself are intended to be self-explanatory. Occasional messages may be produced by the loader(ld).

BUGS

None known, of course.

9.7. Reserved Identifiers

The following identifiers are predefined by the Pascal language.

abs	false	pack	sqr
addr	get	page	sqrt
address	halt	pred	
arctan	integer	put	succ
boolean	ln	read	text
char	mark	readln	true
chr	maxint	real	trunc
convert	minint	release	unpack
cos	new	rewrite	write
eof	nil	round	writeln
eoln	odd	short	
exp	ord	sin	

Further there are identifiers used by the Pascal run time system that are referenced at linking time. Warning: the user might use this identifiers to define own procedures or data and the linker ld will reference the users' entity and report no error.

__entry	_popen	_openfileindx	_pwrb
_pjerr	_environ	_prdr	_peoln
_pwrr	_ppack	_openfiles	_pwrc
_argc	_error	_prdsh	_pget
_plim	_ppage	_pbvec	_pwrc1
_pwrs	_errorexit	_prele	_phigh
_argv	_pperr	_pcase	_pwrcs
_plow	_ferror	_prese	_piabs
_pwrsh	_pput	_pcerr	_pwrf
_closefiles	_final	_prewr	_pierr
_pmark	_prdc	_pclos	_pwri
_syserr	_flushbuffer	_pserr	_pisqr
_copen	_prdi	_pdiv0	_pwrln
_pnew	_localfile	_punpa	
_endoffile	_prdln	_peof	

10. REFERENCES

[1] Jensen K.; Wirth N.: *Pascal User Manual and Report*
Second Corrected Reprint of the Second Edition 1978, Springer Verlag

[2] Addyman, A. M.: *BSI / ISO Working Draft of Standard Pascal*
BSI DPS/13/4 Working Group, Pascal News #14, January 1979

[3] Wichmann A.; Sale J.: *Pascal Processor Validation Suite*
Pascal News #16, October 1979, pp 10-141, Pascal User's Group

[4] Krall D.: *Pascal Validation Suite Report*
ZT Software Newsletter, Heft 1 (1981) pp 6.1-9.6, Siemens AG

[5] *UNIX Programmer's Manual; pc(1)*

[6] *UNIX Programmer's Manual; Volume 2; 2,3,6,...*

[7] Kernighan B.; Ritchie D.: *The C Programming Language*
Prentice-Hall Software Series; Prentice Hall, Inc.; New Jersey

[8] *UNIX Programmer's Manual; prof (1)*

[9] *UNIX Programmer's Manual; adb (1)* and
UNIX Programmer's Manual; Volume 2; 18

[10] *UNIX Programmer's Manual; a.out (5)*

[11] *PERKEO - a Hardware / Software System for Personal, Scientific Computing*
Siemens AG, ZT ZFE FL AIF

[12] *Q68K - Ein Q-Bus-Prozessor hoher Leistung auf 68000-Basis*
Spezification Version 1; PCS interner Bericht; 1982

[13] *UNIX Time-Sharing System*
The Bell Sytem Technical Journal, July-August 1978, Vol. 57, No.6, Part 2

[14] *UNIX Time-Sharing System*

UNIX Programmer's Manual, Seventh Edition, Volume 1 & 2A, January 1979 Bell Telephone Laboratories, Inc.

[15] *UNIX Programmer's Manual; ar(1), ld(1)*

[16] Wiedemann A.: *PBUG ein interaktiver bildschirmorientierter Debugger*
Siemens AG, ZTI INF 212, January 82

[17] Tensi T.; Krall D.: *Pascal Prettyprinter PRETTY*
ZT Software Newsletter, Heft 4 (1979) pp 3.1-3.15, Siemens AG

L E A R N S N O B O L .

Snobol is a programming language with some of the best features of Basic and Lisp:

-- easy to learn

-- 'rubber memory' for strings, lists, associative tables

-- interactive.

These features make Snobol good for :

-- text editing jobs. (EDT is a lousy programming language).

-- small interactive data bases

-- small translators: mini-languages, macros, language front ends for e.g. decision tables, networks, testing.

This introduction is for people who know how to program, and want a quick overview of Snobol. Look at the syntax, study some examples, then try your own.

Snobol runs on the Perkeo Mc8000 Unix, in a very small (32 Kbytes) and fast dialect called 'Spitbol'. (BS2000 has only an ancient and slow Snobol interpreter).

C O N T E N T S .

S T R I N G S .

Strings in Snobol can be assigned, put together, input and output, with little fuss:

```
X       = 'hello'

Hello_Hello = X X

Title   = 'Dear '  Name  ','

OUTPUT  = 'echo: ' INPUT
```

'X = ...' creates a new variable or box named 'X', on the spot; Snobol has no variable declarations.

Strings are put together (concatenated) just by listing them, separated by blanks; thus Hello_Hello is 'hellohello' .

'OUTPUT = ...' writes a line to the terminal, INPUT reads one.

'Quote' and "Quote" are the same. "'" is a single quote, '"' a double quote.

The following Snobol program just echoes each input line, with its size and line number. It uses 'FOR .. NEXT' (described below) to get lines of input:

```
FOR  line = INPUT
        line_nr = line_nr + 1
        OUTPUT = line_nr  ' '  SIZE(line)  ' '  line
        NEXT
```

Numbers are treated in context as strings, and strings as numbers. For example:

```
TWO = '2'
OUTPUT = TWO + TWO
```

does what you would expect.

New variables are initially '', the so-called nil string of length 0. '' + 2 is 2.

Taking strings apart, like 'hello there' --> 'hello', ' ', 'there' is more interesting than putting them together. This is introduced in the section on 'Pattern Matching' .

A R R A Y S and T A B L E S .

```
A = ARRAY( 10 )
```

creates 10 variables, A<1> .. A<10>, which can be used just like ordinary variables:

```
A<2>  =  A<j>  +  INPUT
A<3>  =  'hello'
...
```

```
T = TABLE( 10 )
```

creates an associative table, which is like a telephone book, or an array indexed by strings:

```
Phone = TABLE( 10 )
Phone< 'Jones' > = 123
Phone< 'Smith' > = 456
...
```

Tables have many uses in Snobol: sets, maps, symbol tables. They're very simple and very useful. Every language should have tables.

(Fine points:

Table and array elements can be anything at all: numbers, strings, records, other tables, trees, theorems ... Table indices (keys) can also be anything.

Tables are initially empty, T<x> is nil for all x. Thus:

```
T<x> = T<x>  ' '  num          grows a list of numbers:
T<x> = ''    ' '  num          on the first time thru,
T<x> =       ' '  num  ' '  num2    on the second, ...
```

See the example program Xref below.

Multi-dimensional arrays are declared like:

```
        A = ARRAY( '10,20' )
or:     A = ARRAY( '-10:10,2,3,4' ) .
```

'10' in 'TABLE(10)' is just advice to Snobol; any number of key-value pairs can be put in the table, but 10 is most efficient. TABLE(1000) would waste some space, TABLE(2) some time.

S N O B O L S T A T E M E N T S Y N T A X .

(label) Statement (gotos)

Snobol is line oriented. Statements are written:

one per line:

STATEMENT

or several per line:

S1; S2; S3 ...

or spread over several lines:

LONG STATEMENT
+ in column 1 continues the previous line.

Column 1 is usually blank. But

* in column 1 starts a comment line. With Sniffi,
'--' to end-of-line is also a comment.

LABEL -- non-blank column 1 starts a label.

There are several kinds of statements:

-- normal: assignments, function calls:

```
X = X + 1
X = Func( j, k + 1 )
```

(N.B. Snobol insists on blanks around binary ops !)

-- IF ELSE FI FOR NEXT DEF 'Structured Snobol' statements

-- pattern matching statements use funny operators ? => $. @
described below, for example: line ? pattern => word

G O T O s, F A I L .

Original Snobol has no IF - FOR control flow statements, only GOTOs. First GOTOs will be described, so that you can read original Snobol, and then an IF - FOR preprocessor, which can make programs more readable.

Snobol for 'GOTO Label' is ' :(Label)' after a statement. Labels start in column 1 and end with a blank, not with ':' .

Snobol functions can signal 'Fail' :

```
        func()      :F(fail) S(succeed)
```

is Snobol for 'IF func() fails THEN GOTO fail ELSE GOTO succeed' .

'Fail' is useful for signalling errors:

```
        X = func()      :F(error)
```

and for looping:

```
        FOR  X = generate_next( X )
                        ...
        NEXT
```

Both built-in functions and user functions can 'fail'. For example, the built-in IDENT(x, y) fails if x is not identical to y. User functions say ':(RETURN)' to return, ':(FRETURN)' to signal failure and return. One can think of 'Fail' as an extra wire coming out of the function, beside its data output.

In pattern matching, 'fail' means 'back up, try an alternative'.

Failure terminates a whole statement :

```
        X = IDENT( X, '' )  'default'
```

is Snobol for 'IF X == '' THEN X = 'default'
(ELSE do nothing).

(Fine point: Spitbol but not Snobol has a 'conditional OR', unfortunately with no name; the expression

```
        ( a, b, c ... )
```

evaluates a, b, c ... in turn until one succeeds, which is then the value of the expression. If none succeed, the whole statement fails as usual. Examples:

```
        max = (GT(x,y)  x,  y )
        X = ( try_this(X),  try_that(X),  'too bad.' )
```

Conditional AND is just catenation, as in:

```
        IF  LE( 0, n )  LE( n, 9 )
                ...
```

Structured Snobol: I F, F O R, D E F .

Sniffi is a simple preprocessor of IF, FOR and DEF statements to Snobol GOTOs. (This takes about 2 pages of Snobol).

```
IF  test
        ...
[ ELSE IF  test
        ...

[ ELSE  ...
FI
```

does what you would expect. The word 'THEN' is optional.

```
IF  var ==
value_a :
                ...
value_b :
                ...
[ ELSE  ...
FI
```

does the first block of statements if IDENT(var, value_a), the second if IDENT(var, value_b), and so on. Each case value is on a line by itself, ending with a ':' .

Sniffi also expands 'IF x == y' to 'IDENT(x, y) :F(else)'
'IF x /= y' to 'DIFFER(x, y) :F(else)'
and 'IF no x' to 'IDENT(x, '') :F(else)' .

```
FOR  statement
        ...
NEXT
```

loops until 'statement' signals 'Fail'. This has many uses:

```
FOR  line = INPUT
```

loops thru lines from the standard input file, until end-of-file.

```
FOR  line ? get_word => word
```

loops thru the 'word's that pattern 'get_word' scans from 'line'.

```
FOR  j = 1 .. n
    ...
```

does what you would expect. n may be changed within the loop.

(Many more kinds of FOR are useful in paper programs, but are usually expanded by hand:

```
FOR  word  IN  line
FOR  node  IN  list
FOR  in_node, in_arc  -->  node N in a network
FOR  next_moves( game_board )
```

Sniffi can easily be changed to expand such FORs.

F U N C T I O N S.

```
DEF  func( arg1, arg2 ... )  local1, local2 ...
```

starts a function definition, which ends at the next DEF func or DEF (blank). Snobol defaults all variables to global, except those in DEF local lists. Locals are initially nil. Sniffi expands DEF to a Snobol DEFINE, plus the jump around the function body needed in straight Snobol, plus a ':(RETURN)' at the end.

To return a value, assign it to the 'func' name. To return, say ':(RETURN)'; to signal failure, ':(FRETURN)' .

Example:

```
DEF  fib( n )
        IF  LE( n, 3 )
        THEN    fib = n
        ELSE    fib = fib( n - 1 )  +  fib( n - 2 )
        FI
```

Example:

```
DEF  centre( string, width )  nspace
        nspace = (width - SIZE(string)) / 2
        centre = DUPL( ' ', nspace )  string
DEF  tower( n )  j
        FOR j = 1 .. n
                stars  = DUPL( '*',  2 * j - 1 )
                output = centre( stars, 80 )
        NEXT
```

Example (using the global pattern 'get_word' defined below) :

```
DEF  change_names( line )  word, c, to_word
**      FOR  words in line:
**      if word is in change_table,  change it.
        FOR  line ? get_word => word
                to_word = change_table< word >
**                      is word in the change_table ?
                IF  to_word  /=  ''
                        line ? word  =  to_word
                FI
        NEXT
```

(Fine points:

With DEF f(arg1, arg2, arg3), calling f(x) is the same as f(x, '', ''). This is useful for switches and default arguments; f probably starts with

```
arg2 = IDENT(arg2)  default_2
arg3 = IDENT(arg3)  default_3
```

Most arguments are local to functions (call by value), but tables, data records, and files can be permanently changed. There are in fact 'address of' and 'arrow' operators, but they're seldom used:

Snobol:	.var	$var
C:	&var	*var
PL/I:	ADDR(var)	var->

DATA Structures.

```
DATA( 'Pair(L,R)' )
```

declares a data structure or record type with fields L and R. Then

```
apair = Pair( 'left', 42 )      -- creates a new pair,
                                -- and initializes it.
R(apair) = R(apair) + 1         -- increments the 42 to 43.
L(apair) == 'left'              -- is true.
R(apair) = Pair( 2, 3 )         -- nested  L 'left',  [ 2, 3 ]
```

Some more data structures, with example uses:

```
DATA( 'date(day,month,year)' )
today = date( 6, 'Nov', 81 )

DATA( 'person(name,department,age,hair_colour,sex)' )
aperson = person( 'John Doe', 'ZTI', 32, 'brown', 'm' )

DATA( 'Auto(make,year,colour,license_nr)' )
colour( my_auto ) = 'yellow'

DATA( 'complex(Re,Im)' )
J = complex( 0.0, 1.0 )
**      (but reals aren't implemented on Perkeo).

DATA( 'cons(car,cdr)' )
DATA( 'list(val,next)' )
v = val(alist);  alist = next(alist)
```

Snobol, like Lisp, doesn't care about the types of fields; they can be numbers, strings, tables, nested records ... (But Snobol does check that DATATYPE(x) == 'Pair' before using L(x)). Thus Snobol depends more than most languages on well-chosen variable, field and function names to make programs readable. Some Snobol programs try to encode everything in long text strings; well-named data structures are better.

DATA annoyingly needs a quoted argument string, with no blanks.

Just as with functions, missing arguments are filled out with nil s. Thus date(6) is the same as date(6, '', '').

```
*--------------- Example data structure:  trees: -----

        DATA( 'new_tree(op,L,R)' )
*                       op : op_type == oneof( '+'  '-'  '=' ... )
*                       L, R :  tree or id or number.
        t = new_tree( '=',  'X',
+                           new_tree( '+', 2, 2 ))
*               -- X = 2 + 2;  see example program Parse.


*               -- A tree-walker, rather Lispy,
*               -- to apply a function to all leaves of a tree:
        DEF  walk( atree, leaffunc )
        IF  DATATYPE(atree) == 'new_tree'
                walk(  L(atree),  leaffunc )
                walk(  R(atree),  leaffunc )
        ELSE
                APPLY( leaffunc, atree )
        FI


*--------------- Example data structure:  stacks: -----

        DATA( 'push(top,pop)' )
*                   top : any;  pop : a stack.
        stak = push( 1 )          -- top(stak) == 1,  pop(stak) == ''
        stak = push( 2, stak )    -- top(stak) == 2
        stak = push( 3, stak )    -- top(stak) == 3
        stak = pop( stak )        -- top(stak) == 2 again
        top(stak) = top(stak) + 10        --  2 to 12
              s = stak
    for_vals  val = top(s)        -- FOR  val  IN  stak
              ...
              DIFFER( s = pop(s))  :S(for_vals)


*--------------- Example data structure:  demons.
```

Associative tables can store properties such as numbers and strings, or can store 'demons': procedures which look around at their context before doing something. For example, some screen editors have 'live' function keys, which can be arbitrary procedures. In Snobol one can simply say:

```
        Action< key >  =  .demon
```

and when 'key' is typed:

```
        APPLY( DIFFER( Action< key >)
```

to call 'demon' (if there is one for that key). This style of data structuring can be interesting.

Basic Pattern Matching.

```
line  ?  pattern
line  ?  pattern1 => word1   pattern2 => word2  ...
line  ?  'change this'  =  'to that'
```

scan a line for a pattern, and may pick off substrings, change matched substrings, and set a 'cursor'. Patterns can be very complex, not to say APLish; I describe only a few basic ones. (Original Snobol writes '$' or '.' for '=>', and writes just 'line pattern' instead of 'line ? pattern').

Patterns are composed of:

```
-- literal strings  'exact match'

-- built-in matching functions:
        ANY(   '...' )   -- one          character  from a set
        SPAN(  '...' )   -- one or more  characters from a set
        BREAK( '...' )   -- up to a      character  from a set
        TAB( n )         -- up to the  n th character
        LEN( n )         -- n characters
        ARB              -- anything
        REM              -- the rest of the line

-- Pname   of a previously defined pattern

-- *Pname  of a forward or recursive pattern

-- sequences:     Pat1  Pat2 ...

-- alternatives:  Pat1 | Pat2 | ...

-- pattern => word  : stores the matched substring in 'word'
```

Some examples of patterns :

Get everything up to a blank or comma, and save it in 'word':

```
        line ?  BREAK(' ,')  =>  word
```

('Up to' would have been a better name for BREAK, and 'Some' better than SPAN).

Find a letter, then get some letters and numbers :

```
        alph   = 'abcdefghijklmnopqrstuvwxyz_'
        get_id = BREAK(alph)  SPAN(alph '0123456789') => id
```

Get everything up to a blank or comma, or if there is none, get the rest of the line:

```
        line  ?  (BREAK(' ,')  |  REM)  =>  word
```

Define a pattern to match 0 or more blanks (SPAN matches 1 or more) :

```
        bl = SPAN(' ')  | ''
```

Match a function or procedure name :

```
        line  ?  bl  ('function ' | 'procedure ') bl  get_id
```

Match prices like $0 $3.99 (and also $3.9999):

```
        price = '$'  SPAN('0123456789')          =>  dollars
+               ('.'  SPAN('0123456789') | '')   =>  cents
```

Match prices like $3.99 exactly:

```
        cent = ANY('0123456789')
        price = '$'  SPAN('0123456789')  =>  dollars
+               ('.'  cent cent   | '' )  =>  cents
```

Match: 'the red' | 'the green' | 'a red' | 'a green' :

```
        line  ?  ('the ' | 'a ')  ('red' | 'green')
```

Alternatives are tried left to right; thus in

```
        'an alternative'  ?  'a' | 'an'
```

'a' matches, but leaves the 'n' hanging. Write 'an' | 'a' .

To match a pattern only at the beginning of a line:

```
                line  ?  TAB(0) a_pattern
```

If the global switch &ANCHOR is 1, all patterns are 'anchored' to match only at the beginning.

Patterns can be BNF-like :

```
            element  =  SPAN(alph_num)
+                    |  '('  *sum  ')'
            product  =  element  (('*' | '/') *product  | '')
            sum      =  product  (('+' | '-') *sum      | '')
```

In practice, I very seldom use BNFs; see the example program Parse below for a real string-to-tree parser.

Cursory.

To implement 'FOR word IN line', one wants to start scanning for a word where the previous one left off. Snobol does this with a 'cursor', which can be pictured as a little arrow pointing just before the next character to be scanned:

```
cursor position 0:   Be  wild  of  tongue  clearly
                     ^
cursor position 2:   Be  wild  of  tongue  clearly
                       ^
```

'@c' in the middle of a pattern means 'set c to this position'. 'FOR word IN line' is then done by:

```
        line  ?  TAB(c)  get_word  @c
```

TAB(c) jumps to cursor position c; get_word gets a word; @c sets c after the word, ready for the next time around.

Another example of '@c' uses the builtin &ALPHABET, which on Perkeo is a string of the 128 Ascii characters in order:

```
        &ALPHABET  ?  BREAK(char)  @order
```

gets a char's order in the Ascii alpnabet, 0 to 127.

```
        &ALPHABET  ?  TAB(n)  LEN(1) => char
```

does just the reverse: gets the n th Ascii character.

Constant patterns are often defined at the top of a program:

```
        alph = ...
        get_word = TAB(*c)   BREAK(alph)   SPAN(alph) => word   @c
                 ...
        c = 0
        FOR  line ? get_word
                 ...
```

A disadvantage of such a get_word is that is uses globals c and word. Settle on a convention, such as c for cursor and get_X to get an X.

After a pattern matches, the matched substring can be replaced :

```
        line  ?  'change this'  =  'to that'


        'be clear'  ?  'clear' => x   = 'not un' x
*                                   — 'be not unclear'


*               — change  'l == r'   -->   'IDENT(l, r)'---
        line ? BREAK('=') => L   '=='   REM => R
+            = 'IDENT('       L   ', '   R   ')'


*       -- delete all the blanks in a line:
*          find 1 or more blanks,  delete them,  loop.
*          SPAN signals  'Fail'  when all the blanks are gone.
squeeze  line ? SPAN(' ') = ''       :S(squeeze)
```

Built-in Functions.

```
IDENT( x, y )  succeeds if x and y are identical,  else fails.
IDENT( x )     is the same as  IDENT( x, '' ),  Lisp  (null x) .
DIFFER( x, y )  is the same as  'not IDENT'.

EQ  NE  LT  LE  GT  GE( m, n )  compare numbers, and succeed or fail.
LEQ  LNE  LLT  LLE  LGT  LGE( m, n )  compare strings lexically,  and
                                      succeed or fail.
        (Note:  IDENT( 5, '5' )  fails,  integer /= string,
        but     EQ(    5, '5' )  converts '5' to 5 and succeeds.
        Thus in tables, T<5> DIFFERs from T<'5'>; for  tables  indexed
        by numbers, write T< +str > or T< CONVERT( str, 'INTEGER' )>.


DATATYPE( x )   is  'INTEGER' for integers,
                    'REAL'    for reals  (not on Perkeo),
                    'STRING'  for strings  (including the nil string),
                    'ARRAY'   for arrays,
                    'TABLE'   for tables,
                    'PATTERN' for patterns
                else the name of a user-defined data type:  see DATA.


CONVERT( x, datatype )  converts if it can, or fails.
CONVERT( a_table, 'ARRAY' )  converts a table to an  n by 2 array.


DUPL( string, ntimes )   duplicates a string  n times:
DUPL( ' * ', 5 )         is ' *  *  *  *  * '.


INPUT(  .in  ,, 'in_file_name' )   opens a file for input,
OUTPUT( .out ,, 'out_file_name' )  opens for output.  Then
        line = in        -- reads a line from 'in_file_name', and
        out = ...        -- writes a line to 'out_file_name' .
        FOR  line = in   -- loops until 'in' fails, on end-of-file.

    The names INPUT, OUTPUT, and on Perkeo TERMINAL, are pre-opened.
    TERMINAL reads and writes without newlines.


APPLY( func_name, args ) calls the 'func' with the  'args'.  This  is
useful  for applying a function to all the elements of a list or tree,
or for Lispy 'demons'.


Snobol system switches and counters have names starting with '&'.  The
most common ones:
    &ANCHOR = 1      -- anchor all pattern matches to start at TAB(0).

    &FTRACE = n      -- trace the next n function calls and returns.

    &TRACE = 20;  TRACE( .x );  TRACE( .y )
        -- trace the next 20 assignments to x and y .
           Later in the run,  &TRACE = 100  would then trace the  next
           100 assignments to x, y, and other TRACE d variables.
           STOPTR( .x )  stops tracing of x.
```

Spitbol	Snobol differences.
fast	very slow
line ? pattern	line pattern
atable[j] . or atable< j >	atable< j >
Upper and lower case Ascii (reserved words lower case)	Upper-case Ebcdic
Perkeo Unix I/O	Fortran formatted I/O
Reals not implemented on Perkeo	Reals

Language elements and built-in functions in Spitbol but not Snobol:

= is a normal binary operator:

```
x = y = atable[ j = j + 1 ]
```

Conditional or:

```
( GT(x,y) x,  y )
```

Error trapping: Setexit

SORT(table) yields a 2-column array.

LPAD(string, width) pads a string with blanks on the left,
RPAD(string, width) pads on the right, for nice columnar output.

The Sniffi preprocessor changes all '=> ' to '$ ' . BS2000 Sniffi deletes all ' ? ', and changes '-' to '_' (because '_' on my BS2000 terminal acts as rubout).

T E S T I N G .

The full Snobol language is available at run-time; variables can be printed, changed, and traced, during a run. The function 'in()' in file d.s (/usr/sar/bz/sno/d.s) reads INPUT lines, and intercepts

```
? immediate Snobol
```

lines starting with '?' for immediate excecution. For testing, put d.s ahead of your program, and change INPUT to in() . Then you can type at run time:

```
? var            -- print a variable
                 -- ( in() prefixes  ' output = var' )
? var = 43
? var = new_tree( '=', 'X', sub_tree )

? DATATYPE(x)    -- if in doubt.
                    (Note that  DATATYPE( '' )  is  'STRING') .

? d( table_or_array )   -- displays all indices and values

? DUMP( 1 )          -- dump all variables

? &FTRACE = 20       -- trace the next 20 function call / returns

? &TRACE = 20;  TRACE( .x );  TRACE( .y )
                     -- trace the next 20 assignments to x and y
```

Acknowledgements.

I thank my colleagues D. Krall, F. Schindler, and U. Weng-Beckmann for their help with this paper.

References.

"The SNOBOL4 Programming Language", R.E.Griswold J.F.Poage and I.T.Polonsky, Prentice-Hall 1971. Auf Deutsch:

"Die Programmiersprache SNOBOL4", Carl Hanser Verlag 1976.

-- THE book on Snobol, by its developers; spends too much time on pattern matching, which makes the language seem very complex and hard to learn.

"A SNOBOL4 Primer", R.E.Griswold and M.T.Griswold, Prentice-Hall 1973.

A History of the SNOBOL Programming Languages, R.E.Griswold, in "History of Programming Languages", ed. R.L.Wexelblat, Academic 1981 pages 601-660.

"The ICON Programming Language (C implementation for Unix)", Coutant C.A., R.E.Griswold and S.B.Wampler, U. Arizona Dept. Computer Science TR 81-4, 1981.

-- cleaner, modernized, seemingly well-implemented successor to Snobol. A big language, not interactive.

"Algorithms in SNOBOL4", J.F.Gimpel, Wiley 1976.

-- many useful programs, but dense, not for beginners.

"MACRO SPITBOL -- a SNOBOL4 Compiler", R.B.K.Dewar and A.P.McCann, Software Practice and Experience vol. 7 pp. 95-114 Jan 1977.

-- A fast and compact compiler/interpreter, written in a machine-independent macro assembly language.

S N O B O L examples.

Many: expand ... <red green blue> ...

Turtle: draw patterns on an Owl screen

Kaffee: a little data base

Trenn: hy-phen-a-tion

Ineq: show e.g. ' x <= y < z+3 ' on a screen

Baum: caller -> called function tree

Fortrace: source-level trace Fortran programs

Mac: minimal macro expander

e: simple line-oriented editor

Xref: cross-reference lister

```
* 22 Okt  many.snif   ... <a b c>  <x y z>   -->   ... ax ay az bx by bz cx cy cz

* to generate many  e.g.  compiler test cases  or  Rubik's cube moves,
* expand   ... <a b c> ...   -->   ... a ...
*                                  ... b ...
*                                  ... c ...
* for example:
* if  <short int unsigned int long>  < <= == /= > >=  <short int 0 1>
*   -->  all 4 * 3 * 4 cases:   if short  <  short
*                                  ...
*                               if long  /=  1


 def  many( line )  L, list, x, R, c

 if line ?  break('<') => L   '<' break('>') => list   '>'  rem => R
        list ? tab(0) span(' ') @c
        list = list ' '
        for  list ? tab(c) break(' ') => x  span(' ') @c
                many( L x R )
                next
 else   output = line
 fi
 def
```

```
* 23 Okt   turtle.s   draw simple patterns:  Pen ***, Left 5  Down 6 ...
** -copy  d.s owl.s

* draw simple patterns  a la LOGO:
* (really needs a raster graphic terminal  & joystick)

* cursor left, right, up, down  (number) :
*                       draw  (number times)  with current pen

* Pen | P  string :
*                       draw with 'string',  e.g.  Pen *  or  Pen hello
*                       (does  Pen <nil>  work like  no pen ?)

* Go | G  line column

* Clear | C
*.....................................................................


* init:
        pen = '*';   back = Cleft
* patterns:
        bl  =  span(' ') | ''
        o9  =  span(' 0123456789')
        get_num  =  bl  span('0123456789') => num   | ''
        get_pen  =  bl  (break(',') | rem) => pen

*---------------------------------------------------------------------
 def  draw( line )  c
    num = 1

 if line ? tab(c) +
**                  — big case:  arrow  /  Pen  /  ...
 Cright get_num @c :
                    terminal = dupl( pen, num )
 Cleft  get_num @c :
                    terminal = pen back  dupl( back  pen back,  num - 1 )  back
 Cup    get_num @c :
                    terminal = pen back  dupl( Cup   pen back,  num - 1 )  Cup
 Cdown  get_num @c :
                    terminal = pen back  dupl( Cdown pen back,  num - 1 )  Cdown

 ('pen' | 'p')  get_pen  @c :
                          back = dupl( Cleft, size(pen))

 ('go' | 'g')  o9 => line  o9 => col  @c :
                    terminal = cursor_to( line, col )

 ('clear' | 'c') @c :
                    terminal = Clearwin
 else
                    terminal = '?'      :(freturn)
 fi
 line ? tab(c) span(' ,') @c
 line ? tab(c) rpos(0)       :s(return) f(draw)
*       — loop thru com1, com2 ... separated by blanks and commas
 def
```

```
** 8 Dez   snobol:   Kaffee / Tee  Abrechnung:   kaffee.liste  -->  kaffee.aus
**      (sample output below).
**      bzowy  ZTI INF 212.
**.....................................................................
        name = array( 30 )
        cups = array( 30 )
        paid = array( 30 )
**          a pattern  to scan  0-9  and .  --
        num = span('0123456789.')

**          define a function:  no_dot( '1.23' )  =  123 --
        def  no_dot( price )
             no_dot = price;   no_dot ? '.' = ''

**                        DM( 123 )  = 'DM 1.23'  --
        def  DM( pf )
               pf = le( 0, pf ) le( pf, 9 )  '0' pf
               pf ? rpos(2) = '.'
               DM = 'DM ' lpad( pf, 5 )
        def

**---------------------------------------------------------------------
** ------------ read the  'Name  1.5' ...  list ----------------------
**---------------------------------------------------------------------
        output = '              Anzahl Tassen   [schon bezahlt]  ?'
        input( .liste,, 'kaffee.liste' )
        for  line = liste
               n = n + 1
               line ?  break(' ') => name[n]
+                       span(' ')    num => cup_size
               terminal = rpad( name[n], 12, '.' )
               if (in = input)  /=  ''
                      in ? num => ncups   rem => pd
                      cups[n] = no_dot( cup_size ) * ncups
                      paid[n] = no_dot( pd )
                      total_cups = total_cups + cups[n]
               fi
        next

**---------------------------------------------------------------------
** ------------ compute each person's cost ---------------------------
**---------------------------------------------------------------------
        terminal = 'Gesamt Ausgabe .';  Gesamt = no_dot( input )
        output( .aus,, 'kaffee.aus' )
        aus = dupl( '=', 78 )

        for j = 1 .. n
        if  cups[j]  /=  ''
               dm = cups[j] * Gesamt / total_cups
               aus = rpad( name[j], 14 )
+                    lpad( cups[j] / 10, 3)  ' Tassen  -->  '
+                    DM( dm - paid[j])
               aus = dupl( '-', 78 )
               total_dm = total_dm + dm
        fi
        next
end
```

Sample output of the -- Kaffee -- program:

```
================================================================================
Brix            48 Tassen  -->  DM  7.29
--------------------------------------------------------------------------------
Bzowy          135 Tassen  -->  DM -2.05
--------------------------------------------------------------------------------
Heinemann      120 Tassen  -->  DM  8.38
--------------------------------------------------------------------------------
Keller          62 Tassen  -->  DM  9.42
--------------------------------------------------------------------------------
Muehlmann        5 Tassen  -->  DM   .79
--------------------------------------------------------------------------------
Petersen         1 Tassen  -->  DM   .22
--------------------------------------------------------------------------------
Schindler      153 Tassen  -->  DM 13.26
--------------------------------------------------------------------------------
Schuchmann       3 Tassen  -->  DM   .45
--------------------------------------------------------------------------------
Weng-Beckmann   31 Tassen  -->  DM -16.72
--------------------------------------------------------------------------------
Wiedemann       21 Tassen  -->  DM -9.88
--------------------------------------------------------------------------------
Student-1        8 Tassen  -->  DM  1.21
--------------------------------------------------------------------------------

Preis pro Normtasse  =  DM   .15     =  DM 89.65  /  590
================================================================================
```

```
* trenn.snobol  26 Nov

**  Trenn rules:

**      v-v     except ei ie au ...
**      v-Kv    K a consonant  or ch sh sch ...
**      k-kv    except            ch sh sch ...

**      disallowed:  -bl  b-l  bl-    -ng n-g gn-

**      keep first 3 or 4 letters together ?
-nolist

*-------------------------------------------------------------------------
  Kblock = 'ch' | 'st' | 'ph' | 'th' | 'sh'
  Kblock3 = 'sch'
  Xblock = 'ng' | 'bl' | 'pl' | 'dl' | 'tl' | 'gl' | 'kl'
+               | 'br' | 'pr'        | 'tr' | 'gr' | 'kr' | 'gn'

  Ublock = 'ei' | 'ie' | 'au' | 'eu' | 'ae' | 'oe' | 'ue' | 'ui' | 'oi'
  Ublock3 = 'aeu'

  Rules = 'VV'  | 'UV'  | 'VU'
+       | 'VKV' | 'UKV' | 'VKU'
+       | 'VCCV' | 'UCCV' | 'VCCU' | 'UCCU' | 'VCCCV' | 'KCCCV'
+       | 'KKV' | 'KKU'

  &anchor = 0

*-------------------------------------------------------------------------
  def  trenn( word )  c
  code = replace( word, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
+                       'abcdefghijklmnopqrstuvwxyz' )

  if   code ? tab(2) any('aeiou')  =  'UUU'
  else code ? len(3)               =  'CCC'
**                      ==> first 3 or 4 letters  stay together ?
  fi
  for code ? Kblock  = 'CC'
  next
  for code ? Kblock3 = 'CCC'
  next
  for code ? Ublock  = 'UU'
  next
  for code ? Ublock3 = 'UUU'
  next
  for code ? Xblock  = 'XX'
  next
  code = replace( code, 'abcdefghijklmnopqrstuvwxyz',
+                       'VKKKVKKKVKKKKKVKKKKKVKKKKK' )

  c = 2
*                       VV  -->  V-V  usw.  --
  for code ? len(c) @c  Rules
      code ? pos(c + 1)  =  '-'
      word ? pos(c + 1)  =  '-'
  next
  trenn = word
  def
```

```
* Mittwoch 29 Juli   i n e q . s :  parse row of inequalities,  2D out :
*          x <= y < z - 2   -->     xxxxxxxxxx
*                                              yyyyyyyyyy . . zzzzzzzzzz
*
*   ineq_in:   parses  ' x +-n  <= y+-n  <  ... '   -->   ivar, inum
*   ineq_out:  ivar, inum   -->  2-dimensional output.

*.....................................................................

*   patterns --
  name = span( '_0123456789abcdefghijklmnopqrstuvwxyz'
+                       'ABCDEFGHIJKLMNOPQRSTUVWXYZ' )
  num  = any( '+-' ) span( '0123456789' )  | ''
  get_var_num  =  name => var   num => n   @c

  &alphabet ? tab(10) len(1) $ nl

  ivar = array( 10 )
  inum = array( 10 )

*---------------------------------------------------------------------
*   ' x +-n  <= y+-n  < z+-n ... '      -->     ivar, inum< 1..nvar >
*---------------------------------------------------------------------
  def  ineq_in( line ) c, n
    line = no_blank( line )
    line ? get_var_num
    ivar[1] = var
    inum[1] = +n
    nvar = 1
    for  line ? tab(c)  ( '<=' | '<' ) => lt   get_var_num
        n = ( ident( '<', lt )    n - 1
        nvar = nvar + 1
        ivar[nvar] = var
        inum[nvar] = +n
    next


*---------------------------------------------------------------------
*          x <= y < z     -->     xxxxxxxxxx
*                                              yyyyyyyyyy zzzzzzzzzz
*---------------------------------------------------------------------
  def  ineq_out() j, line, gap
    for j = 1 .. nvar
        line = line  dupl( ivar[j], 10 )
        gap = inum[j + 1] - inum[j]
        line = line  ( eq( gap, 0 )  ' ',
+                    gt( gap, 0 )  dupl( ' . ', gap ),
+                                  nl  dupl( ' ',  size(line) + gap )
    next

  def
```

```
* 20 Okt  baum.snobol     call tree:  proc A   -->  B, C  -->  C1, C2 ...
*
*   build the proc / call tree of a pl/m program (functions ?),
*   print it out nicely indented:
*       proc A  -->  B
*                    C  -->  C1 C2 ...
*
    ein      = table( 100 )
    aus      = table(  50 )
    level    = table(  50 )
    schon_aus = table( 100 )
    out_linum = 1

    &anchor = 0
    input( .read );  &trim = 1;
    output( .out )
    bl = span(' ') | ''


 def  baum( proc, einrueck )  sohn, soehne
    soehne = aus[ proc ]
    if == soehne
        out = einrueck proc
        out_linum = out_linum + 1

    else if == schon_aus[ proc ]
        schon_aus[ proc ] = out_linum
        out = einrueck proc
        out_linum = out_linum + 1
            soehne ? ' ' = ''
        for soehne ? break(' ') => sohn  ' '  =  ''
            baum( sohn, einrueck '     ' )
        next
    else
        out = einrueck proc ' ... ' schon_aus[ proc ]
        out_linum = out_linum + 1
    fi


 def  add( elt, list )
*           -- encode list as ' a b c ... '  long string
    if == $list
            $list  =  ' ' elt ' '
    else if $list  ?  ' ' elt ' '
    else    $list = $list  elt ' '
    fi

 def  max( m, val )
    $m = lt( $m, val )  val

 def
*
```

```
*-------------------------------------------------------------------------
    for line = read

    if      line ? bl  'proc ' bl  break(':')   => proc
    else if line ? bl  'call ' bl  break('( ;') => call

*------------------------- link  proc  -->  call -------------------------
        add( call,  .aus[ proc ])
        add( proc,  .ein[ call ])
*-------------------------------------------------------------------------

        max( .level[ proc ],  level[ call ] + 1 )
    fi
    next

*                   first put out:   proc   --> aus   <-- ein
    ein_array = convert( ein, 'array' )
    ja = 1
    for proc = ein_array[ ja, 1 ]
        ja = ja + 1
        out = level[proc]  rpad( proc, 12 )
+           '  -->  ' aus[proc]
+           '  <--  ' ein[proc]
    next
        out = '';  out = dupl( '-', 79 );  out = ''


    output = 'main procedure ?';  root = input
    root = ident(root)  proc

    baum( root, '' )

end
```

```
* 20 Okt  fortrace.snobol   source-level trace of a fortran run
* bzowy  (retyped from version 22 Feb)

* Trace all statements in a Fortran program, by expanding
*   assignments:    X = ...    -->
*                   call trace( 'X = ... text',  X )

*   ifs, calls, dos:    call subr(...)   -->
*                       call trace( 'call subr(...) text',  no_val )
*                       before the if / call / do

*   'Trace' writes its text arg 1,  and arg 2 if /= no_val  (G format ?.
*   and could also read  'Nr. trace lines more ? '  like snobol &TRACE .

*   (Of course this technique doubles the size of the source,  but so what ?


    &anchor = 1
    input( .read, 5 );  &trim = 1
    output( .write, 3 )


 def  ftrace( line, var )  c, long
    if /= subr_line
        long = subr_line;  subr_line = '';  ftrace( long )
*       -- function | subr line;  it would be nice to write the arguments.
    fi
    var = ident(var)  -4081
*       -- ftrace( text, -4081 )  =>  text only, no val
*       -- (overloads ftrace,  better 2nd entry point ?
    line ? tab(0) '      ' = ''
*-------------------------------------------------------------------------
    line = "X        call  trace( '" line "', " var ")"
*-------------------------------------------------------------------------
    line ?            (len(72) | rem)        => write  @c
    for line ? tab(c) (len(66) | len(1) rem) => long   @c
        write = '     *' long
        next
 def

*
```

```
*---- main loop: ------------------------------------------------------
read    line = read                                             :f(end_in)
        line_nr = line_nr + 1

*               -- c/notrace ?  necessary around 1-line functions --
        line ? 'c/' rem => notrace                              :s(out)
        ident( notrace, 'notrace' )                             :s(out)

        line ? span(' 0123456789') @c                           :f(out)
        line ? '     ' notany(' ')                              :s(out)
*               -- fortran continuation line ?  (free-form ?


*---------- ftrace call | do before,  assign after ------------------------

        if line ? tab(c) +

        'call ' | 'do ' | 'if ' | 'if(' | 'go ' | 'goto ' | 'return' :
                ftrace( line )                                  :(out)

        'format' | 'data' | 'parameter' :
                                                                :(out)
        'subroutine ' | 'function ' :
                subr_line = line                                :(out)
*                       -- hold until common, int, data ... are out

        break( '="' "'" )  => var  '=' :
                write = line
                ftrace( line, var )
*                       -- trace after 'X = ...'  to print the new X.
*                       -- (look ahead for continuation lines ?
        fi

out     write = line                                            :(read)
end_in
end
```

e: a little line editor.

---------- moving ----------

n print n lines

+n go down n lines

-n go up n lines

<return> go to, and print, the next line

Ln go to line n. L1 goes to the first line, L-1 to the last.

L what line am I on?

/find find the next line containing f i n d
/find/ a second / is optional,
but is needed in, for example: /find/-1 3

/ repeat the last /find

?find find the previous line containing f i n d

---------- changing ----------

I
insert lines before the current line;
to stop inserting, type a single
.

Kn kill (delete) n lines

C/this/that Change: find /this/, and change it to /that/
C repeat the last change
S for Substitute, is the same as C

A/str/ ... All: do the commands ...
for all lines containing /str/
G for Global, is the same as A

R filename read a file,

W filename write a file.

Q quit.

```
* 27 jan  e.snif:   a little line editor,  described in  e.man

**          — data structures,  globals,  patterns —

        data( 'new_line(text,Lprev,Lnext)' )
**              — a 2-way linked list of all lines
        L0 = new_line()
**              — line 0:  before first,  after last
        Lprev(L0) = Lnext(L0) = L0
        L = L0
**              — global:  current line

        num      =  span('0123456789') => n
        opt_num  =  span('0123456789') => n  | ''

        find_pattern = any('/?')       => slash
+                     (break(*slash) => str  len(1)     |  rem => str )

        change_pattern = '/'  break('/')  => this
+                         '/' (break('/')  => that  '/'   |  rem => that )
        &anchor = 0

*-------------------------------------------------------------------------
 def  insert1( text )
        Lprev(L)  =  Lnext(Lprev(L))  =  new_line( text,  Lprev(L),  L )

 def  insert( file, endfile )  line
        for (line = $file)  /=  endfile
                insert = insert + 1
                insert1( line )
        next

*-------------------------------------------------------------------------
 def  write( file, n )
**                      — leaves L at  n th  line,  or L0
        $file = text(L)
        write = write + 1
        gt( n = n - 1,  0 )             :f(return)
        differ( L = Lnext(L),  L0 )     :s(write) f(return)

*-------------------------------------------------------------------------
 def  find( str, up )
        if no str
                str = find_last
        else    find_last = str
        fi
find1   text(L) ? str                   :s(return)
        L = (ident( up, '?')  Lprev(L),  Lnext(L))
        differ( L,  L0 )                :s(find1) f(freturn)

*-------------------------------------------------------------------------
 def  down( n, up )
        L = (ident( up, '-')  Lprev(L),  Lnext(L))
        gt( n = n - 1,  0 )   differ( L,  L0 )  :s(down) f(return)
```

```
*=============================================================================
**              — e  page 2:  for commands in opline:

 def  e( opline )  c
        opline = ident(opline)  '+1'
eloop
    n = 1
    opline ? tab(c) span(' ') @c

 if opline ? tab(c) +

 num @c:
                write( .output, +n )
                opline ? tab(c) rpos(0)          :s(return)
 rpos(0):
                write( .output, 1 )              :(return)
 '+' opt_num  @c:
                down( +n )
 '-' opt_num  @c:
                down( +n, '-' )

*-----------------------------------------------------------------------------
 'l'  ('-' | '') => up   num   @c:
**              — L1  goes to first line,  L-1  last.
                L = L0;  down( n, up )
 'l'  @c:
                n = 0;        t = L0
 lno            n = n + 1;   differ( t = Lnext(t),  L )  :s(lno)
                output = 'line '  n  ':'

*------------------------------ Find,  Change ------------------------------
 find_pattern  @c:
**              — first advance,  so  f f f ...  finds different lines
                L = (ident( up, '?')  Lprev(L),  Lnext(L))
                find( str, slash )

 any( 'cs' )  (change_pattern | '')  @c:
                find( this )
                text(L) ? this = that

*------------------------------ All  |  Global ------------------------------
 any( 'ag' )  find_pattern  rem => opline :
                opline = ident(opline )  '1'
                find_all = (differ(str) str,  find_last )
                L = Lnext(L0)
                for  find( find_all )
                    e( opline )
                next

*-----------------------------------------------------------------------------
 'i'  rem @c:
                insert( .input, '.' )

 'k'  opt_num  @c:
                k = Lprev(L);   down( n + 1 )
                Lnext(k) = L;   Lprev(L) = k
 **                     — save the killed lines to move, unkill ?
```

```
 'r ' rem => filename  @c:
                 input(  .readfile, .chan, filename )
                 output = insert( .readfile, -1 )  ' lines read.'
                 endfile( .chan )

 'w ' rem => filename  @c:
                 L = Lnext(L0)
                 output( .writefile, .chan, filename )
                 output = write(  .writefile, 99999 )  ' lines written.'
                 endfile( .chan )
 else
                 output = '?  what s "' opline '" ?'     :(freturn)
 fi

 L = ident( L, L0 )  Lnext(L0)
 :(eloop)
def


loop    e( in())  :(loop)
end
```

```
* 17 Juli  Spitbol example:  cross-reference lister
-nolist

  line_nr_table = table( 1000 )
*       a table:  word  -->  list of line nrs:
*       (tables are like arrays,  indexed by strings,  growing as needed)

*       Pattern to get a 'word' --
  letter = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789'
  get_word  =  tab(*cursor)  break(letter)  span(letter) => word
+              @cursor

*               for lines in file,  for words in line --
  for  line = input
       line_nr = line_nr + 1
**     output = line_nr '  ' line
       cursor = 0

  for  line ? get_word
       line_nr_table[ word ] =
+      line_nr_table[ word ]  '  ' line_nr
*          -- just a long string of line_nr s
*          -- (Page# - line#  would be better).
  next
  next

       sorted = sort( line_nr_table )
       w = 0
  for  word = sorted[ w = w + 1, 1]
       output = rpad( word ' ', 12, '.' )   sorted[w,2]
  next
end


------------------------  Sample output:  Xref of itself:  -------------------
0 ..........  17  27
1 ..........  15  28  28
1000 .......  4
12 .........  29
17 .........  1
2 ..........  29
Juli .......  1
Pattern ....  8
Spitbol ....  1
a ..........  5  8  22
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789   9
are ........  6
arrays .....  6
as .........  6
break ......  10
by .........  6
cross ......  1
cursor .....  10  11  17
end ........  31
example ....  1
file .......  13
for ........  13  13  14  19  28
```

PBUG ein interaktiver bildschirmorientierter Debugger

ABSTRACT

PBUG ist ein interaktiver bildschirmorientierter Debugger, der unter dem Betriebssystem UNX† ablauffaehig ist. Mit Hilfe einer Menuetechnik lassen sich C- und PASCAL Programme in einer sehr quellsprachen nahen Form ohne Kenntnis maschinennaher Details testen.

INHALT

†UNX is a Trademark of Bell Laboratories.

Dieser Bericht wurde unter dem Betriebsystem UNIX mit dem Textformatierer NROFF/TROFF aufbereitet. NROFF/TROFF hat nur einen Trennalgorithmus fuer das Englische, der sich beim Deutschen manchmal recht merkwuerdig benimmt.

1. Einfuehrung

Trotz der Bestrebungen, durch Programm-Konstruktion und Verifikation fehlerfreie Programme zu erzeugen, wird fast jeder Benutzer eines Computers, der sich mit dem Erstellen von Programmen befasst, frueher oder spaeter mit dem Auffinden von Programmier- und/oder Tippfehlern konfrontiert. Fuer die Fehlersuche bieten sich drei Moeglichkeiten an:

1) Ein hexadezimaler Post-Mortem-Dump vom Absturz des Programms.
2) Mit Papier, Bleistift und logischem Denken am Schreibtisch, eine manchmal nicht einfache Methode.
3) Durch gesteuerten und ueberwachten Ablauf des Programms und durch Ausgabe, manchmal auch Veraendern, von Variableninhalten mit einem Debugger.

Die erste Methode ist sehr muehsam und zeitaufwendig und entspricht in etwa einer Autopsie. Sie wird oefters den Weg zur Absturzursache nicht preisgeben. Im allgemeinen ist die zweite Methode zur Fehlersuche zu bevorzugen, doch es treten manchmal Fehler auf, die erst mit der dritten Methode schnell und einfach gefunden werden koennen.

Dieser Bericht befasst sich mit einem Debugger fuer die 3. Methode. Es wird auf den Funktionsumfang und die Benutzerschnittstelle eingegangen. Anschliessend wird PBUG ein auf Zweidimensionalitaet ausgerichteter, bildschirmorientierter Debugger auf dem PERKEO-System mit dem MC 68000 unter dem Betriebssystem UNIX vorgestellt.

2. Ueberblick ueber die Funktionen von Testhilfen

Die heute ueblichen Testhilfen lassen sich grob in zwei Klassen einteilen:

1) Monitore
2) Debugger

In diesem Bericht ist unter Monitor ein Testmittel zu verstehen, das es erlaubt Register und Speicherinhalte in einfacher Form anzuzeigen und zu veraendern; Ein Monitor ist vorallem fuer kleine Programme zu verwenden. Er arbeitet auf hexadezimaler Ebene und erfordert von einem Benutzer ziemlich genaue Kenntnisse der Maschinenstruktur und des erzeugten Codes. Monitore erfordern keine Zusatzinformation, da sie nur hexadezimale Muster im Speicher verund bearbeiten.

Debugger sind fuer mittelgrosse Programme geeignet und erlauben teilweise das Arbeiten auf symbolischer Ebene. Sie unterstuetzen auch das Testen von Programmen in einer HLL. Bei Debuggern, vor allem wenn sie symbolisches Debuggen erlauben, sind einige Zusatzinformationen erforderlich, die das Testprogramm oder das Betriebssystem bereitstellen muss.

Im folgenden wird auf Debugger eingegangen, da Monitore einen sehr eingeschraenkten Verwendungsbereich haben.

Beim Debuggen erfolgt der Ablauf des zu testenden Programms (im folgenden Testprogramm) unter Kontrolle und Steuerung eines Debuggers. Dieses kann man einerseits durch gezieltes Setzen von Haltepunkten an beliebigen Stellen des Testprogramms erreichen, andererseits besteht die Moeglichkeit der schrittweisen Abarbeitung oder der Unterbrechung. Verfeinert werden kann der gesteuerte Ablauf bzw. das Halten durch Bedingungen.

Einige Debugger koennen einen symbolischen Post-Mortem-Dump erzeugen. Dieser enthaelt ueblicherweise die Aufrufhierarchie der Prozeduren, die meistens durch die Angabe der Zeilennummern, in der die Aufrufe stehen, ergaenzt wird. Damit laesst sich die Absturzstelle des Programms ziemlich genau lokalisieren. In manchen Faellen werden auch die Aufrufparameter und die prozedurlokalen Variablen angezeigt (siehe Bild 1).

```
~look_up_symb(#0080,#0736,#0080,#0722) line 123
            form_curr_nr:    #00800736
            form_curr_offset: #00800722
            loc_index:       #056e0000

~look_up_mod_nam() line 160

~pbmain() line 30
            loc_char:        #0d000000

~_main() line 53
```

Bild 1: Vom Post-Mortem-Dump ausgegebene Aufrufhierarchie der Prozeduren

Nachdem der Ablauf eines Testprogramms mit einer der eben angegebenen Methoden abgebrochen wurde, nimmt der Debugger Kommandos entgegen. Ueblicherweise wird die Ausgabe von Speicherinhalten oder Registern angefordert. Komfortablere Debugger erlauben eine Formatierung der Ausgabe. Haeufig muss der gewuenschte Speicherbereich durch eine hexadezimale Adresse angegeben werden, die muehsam aus einem Binderlisting zu entnehmen ist. Wuenschenswerter ist jedoch die symbolische, moeglichst der Programmiersprache angepasste Form.

Einige Debugger bieten die Moeglichkeit den Maschinencode in mnemotechnischer Assemblerschreibweise auszugeben, was das manuelle Decodieren des Maschinencodes erspart.

Ein Aendern von Daten, teilweise auch des Codes mit dem Debugger ist haeufig zulaessig, doch sollte von dieser Methode wegen ihrer moeglichen Seiteneffekte sehr vorsichtig Gebrauch gemacht werden.

Diese Aufzaehlung von Debugger Funktionen gibt nur einen Ueberblick und erhebt keinen Anspruch auf Vollstaendigkeit.

3. Vorteile einer bildschirmorientierten Ein/Ausgabe

Soweit dem Autor bekannt, arbeiten Debugger im Dialog meist auf der Zeilenebene, d.h. nach Eingabe einer Kommandozeile erfolgt die gewuenschte Ausgabe in ein oder mehreren Zeilen, danach kann wieder eine Eingabesequenz erfolgen usw..

Bei heutigen Kleinrechnern und Personal Computern sind die Bildschirmgeraete im allgemeinen ueber schnelle Datenleitungen direkt am Rechner angeschlossen, so dass eine zeichenweise Ein-/Ausgabe keine wesentlichen Probleme aufwirft. Unter dieser Voraussetzung kann auch die Cursorpositionierung von der Tastatur aus Verwendung finden. Auf diese Weise lassen sich Korrekturen am Eingabetext oder Markierungen am ausgegebenen Text einfach bewerkstelligen. Ausserdem koennen Eingaben in Abhaengigkeit von der Cursorposition unterschiedlich ausgewertet werden. Dies bedeutet, dass der Bildschirm nicht nur als eindimensionales Medium verwendbar ist, sondern als Flaeche, die unter Umstaenden in mehrere Bereiche fuer verschiedene Ausgabeinformationen unterteilt sein kann.

Der Vorteil einer derartigen Anzeigeform besteht darin, dass wichtige Informationen beliebig lange auf dem Bildschirm zu sehen sind, waehrend weniger wichtige ueberschrieben werden koennen. Es liegt nahe, fuer ein derartiges Vorhaben verschiedene "Fenster" auf dem Bildschirm vorzusehen.

Im Prinzip koennen sich diese Fenster ueberlappen oder es koennen mehrere Fenster uebereinander liegen und nur das jeweils aktuelle ist mit seinem Inhalt sichtbar. Um groessere Informationsmengen in einem solchen Fenster darzustellen, bietet sich ein Mechanismus an, der ein Verschieben des Fensters ueber dem Ausgabetext zulaesst, um so verschiedene Bereiche sichtbar zu machen. Diese Methode wird auch "Blaettern" genannt.

Ein weiterer Vorteil der Fenstertechnik besteht darin, dass verschiedenartige Informationen in verschiedenen Fenstern angezeigt werden koennen, so dass ein Benutzer schon daran erkennen kann, ob es sich z.B. um Fehlermeldungen, Statusanzeigen, Speicherinhalte oder aehnliches handelt. Diese Methode erleichtert einem Benutzer das Arbeiten mit einem Programm, da z.B. fehlerhafte Eingaben direkt durch den Cursor angezeigt und korrigiert werden koennen.

Ein Nachteil der Fenstertechnik ist in der etwas staerkeren Abhaengigkeit von den Ein-/Ausgabegeraeten zu sehen.

4. PBUG ein bildschirmorientierter Debugger

Die fuer einen Benutzer komfortabelste Methode beim Programmtesten besteht darin, dass einerseits das Testen eines Programmes auf der Quellsprachenebene vorgenommen wird und andererseits die Zweidimensionalitaet des Bildschirms fuer dieses Testen ausgenutzt wird. Bisher ist diese Methode bei sehr wenigen Debuggern und wenn dann nur in eingeschraenkter Form vorzufinden.

Dies war fuer den Autor der Anlass, dieser Idealvorstellung durch Entwicklung eines entsprechenden Debuggers moeglichst nahe zu kommen. Im folgenden wird die Implementierung eines Prototyps kurz beschrieben. Diese vorlaeufige Loesung ist unter dem Programmnamen PBUG (Perkeo Debugger) auf dem Perkeorechner ablauffaehig.

Im folgenden wird das Programm PBUG vorgestellt. Dabei wird sowohl die Benutzerschnittstelle, als auch der Aufbau des Programms erlaeutert.

4.1. Benutzerschnittstelle

PBUG verwendet die oben genannte Fenstertechnik. Dabei wird der Bildschirm wie in Bild 2 zu sehen aufgebaut.

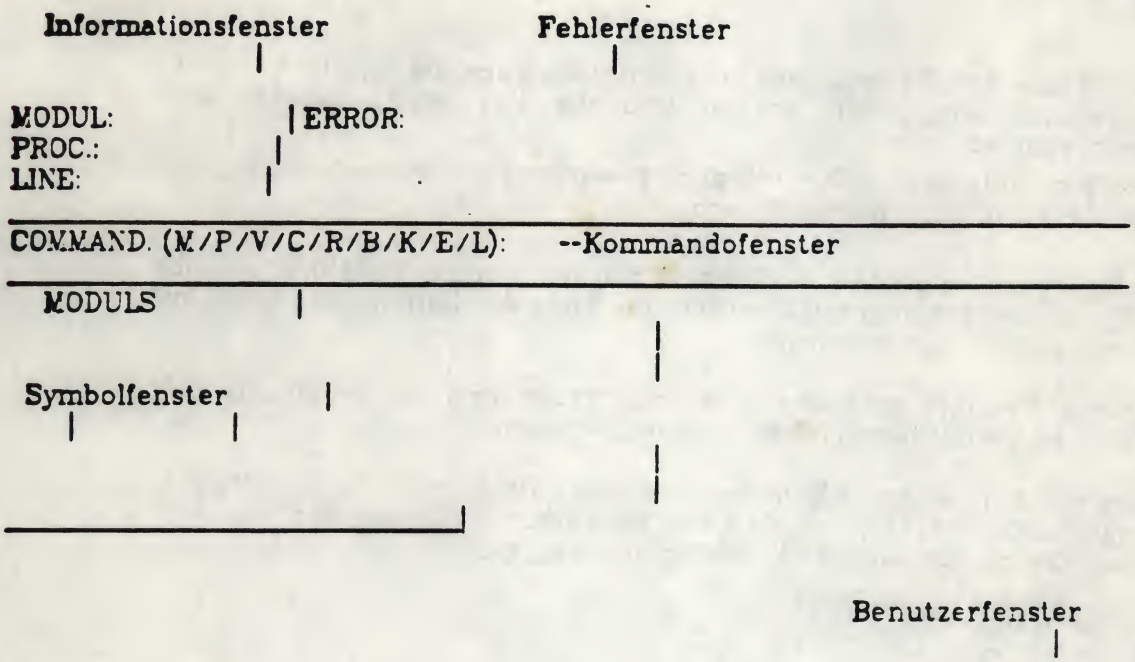


Bild 2: Aufteilung des Bildschirms bei PBUG

Im Informationsfenster erscheinen der jeweils aktuelle Modul- und Prozedurname sowie die aktuelle Zeilennummer des Quellprogramms. Programm- und Eingabefehler werden im Fenster fuer Fehlermeldungen angezeigt. Im Kommandofenster erscheint entweder ein Menue mit den Funktionen des Debuggers, aus denen eine ausgewaehlt werden kann, oder eine kurze Beschreibung der jeweils zulaessigen Eingaben. Im Symbolfenster koennen Speicherinhalte und Symbolnamen des Testprogramms aufgelistet werden.

In den folgenden Abschnitten wird die Ablaufsteuerung des Testprogramms und die Ausgabe von Daten kurz beschrieben.

PBUG erlaubt in seiner derzeitigen Ausbaustufe das Setzen von Haltepunkten sowohl am Anfang als auch am Ende von Prozeduren. Das Testprogramm kann, mit Parametern versorgt, gestartet werden und haelt am ersten Haltepunkt an. Fuer den Ablauf des Testprogramms von Haltepunkt zu Haltepunkt steht eine

Continue-Funktion zur Verfuegung. Beim Erreichen eines Haltepunktes koennen weitere Haltepunkte gesetzt oder geloescht werden und Variableninhalte angezeigt werden. Das Testen des Programms kann selbstverstaendlich jederzeit beendet werden.

Das Markieren von Haltepunkten und Variablen kann auf zweierlei Art geschehen:

1) Der Name der Prozedur oder der Variablen kann als Text im Kommandofenster eingegeben werden und die Markierung gesetzt oder geloescht werden.
2) Im Symbolfenster kann bis zum gewuenschten Symbolnamen positioniert und dann markiert werden.

An einem Haltepunkt koennen die Namen der markierten Variablen und ihr Inhalt im Symbolfenster angezeigt werden. Die Liste der Haltepunkte kann auf dieselbe Weise zur Anzeige kommen.

Prozedurlokale Variable sind ebenfalls markierbar und am jeweiligen Haltepunkt koennen sie mit ihrem Inhalt angezeigt werden.

Die Ausgabe von Daten kann z.Z. in verschiedenen Datentypformaten erfolgen. Der voreingestellte Datentyp ist 4 Bytes hexadezimal. Datentypen, die vom Benutzer per Kommando eingestellt werden koennen sind:

- "X": 4 Bytes hexadezimal
- "x": 2 Bytes hexadezimal
- "D": 4 Bytes dezimal
- "d": 2 Bytes dezimal
- "B": Booleanwerte TRUE/FALSE
- "C": ASCII-Zeichen mit Ersatzdarstellung fuer Steuerzeichen
- "S": Sets als Binaermuster
- "^": Dereferenzierung

Datenstrukturen koennen behelfsmaessig in verschiedenen Datentypformaten angezeigt werden. Dabei ist auch eine Dereferenzierung moeglich.

Aufruf:

mktree <progname> — erzeugt Symboltabelle <progname.sym>

plug <progname>

Hinweis: bei VT100 bzw. PCS DSG entspricht "esc" der Taste "enter"
(ctl-S : Bildschirm-Refresh)

4.2. Aufbau

PBUG ist als ein Zustandsautomat mit einem zentralen und mehreren peripheren Zustaenden aufgebaut. Vom zentralen Zustand kann in die einzelnen Programmroutinen verzweigt werden (siehe Bild 3).

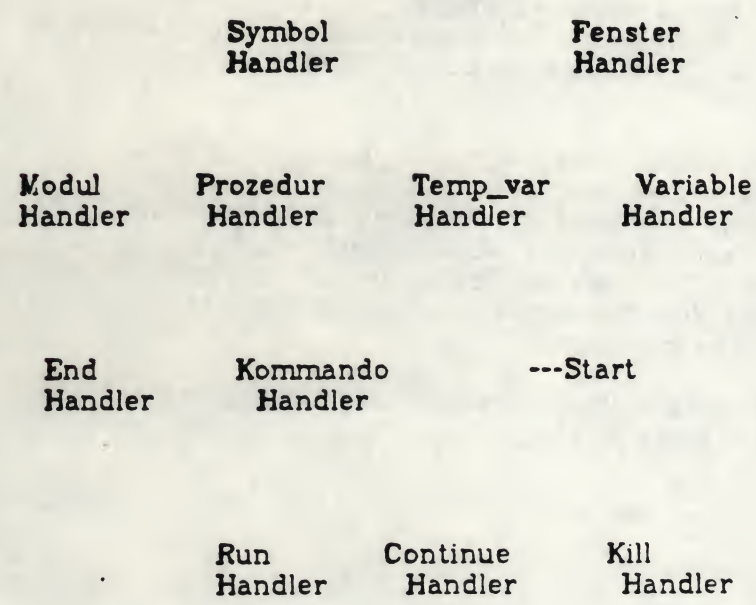


Bild 3: Zustandsdiagramm von PBUG

Jede dieser Routinen stellt ein aus mehreren Modulen aufgebautes Programmstueck dar, das eine bestimmte Funktion realisiert. So gibt es z B. einen Variablen-Handler, einen Prozedur-Handler, einen Modul-Handler usw. und jeder dieser Handler laesst bestimmte Eingaben zu, die entsprechende Ausgaben veranlassen. Beim Prozedur-Handler z.B. koennen Haltepunkte gesetzt, geloescht und angezeigt werden, beim Variablen-Handler ist die Ausgabe von Variableninhalten in verschiedenen Datentypformaten moeglich.

Manche dieser Routinen haben gemeinsame Funktionen. Diese werden, soweit es moeglich ist, von zentralen Prozeduren ausgefuehrt.

5. Zusammenfassung:

Bei PBUG wurde der Versuch unternommen, die meist bei Debuggern vorzufindende Zeilenorientierung zu umgehen und stattdessen die Zweidimensionalitaet des Bildschirms auszunutzen.

Mit ein Ziel des Debuggers war es, soweit wie moeglich jede Maschinenabhaengigkeit vom Benutzer fernzuhalten. Mit Hilfe von PBUG lassen sich sowohl C- als auch Pascal-Programme testen und auch gemischte C-Pascal-Programme bereiten keinerlei Schwierigkeiten.

In der derzeitigen Ausbaustufe von PBUG ist es noch nicht moeglich, in jeder Zeile einer Prozedur Haltepunkte zu setzen, ebenso koennen komplexere Datenstrukturen nur behelfsmaessig angezeigt werden. Dies und das Verarbeiten von Datentypen einzelner Variablen sind in Arbeit. Unser entfernteres Ziel ist es, dem Benutzer direkt den Quelltext des Testprogramms anzuzeigen und ihm die Moeglichkeit zu geben, dort mit dem Cursor Haltepunkte festzulegen sowie Variable zur Ausgabe zu kennzeichnen.

Solange Programme fehlerbehaftet sind, sollten benutzerfreundliche Debugger zur Verfuegung stehen. PBUG zielt in diese Richtung.

6. Literatur:

[1] J.F. Maranzano, S.R. Bourne *A Tutorial Introduction to ADB* UNIX TIME_SHARING SYSTEM. UNIX Programmers Manual Seventh Edition Volume 1 Bell Telephone Laboratories, Incorporated Murray Hill, New Jersey

[2] *MACSBUG* Motorola Design Module User's Guide MEX68KDM(D3)

[3] N. Wirth. *The Personal Computer LILITH* Eidgenoessische Technische Hochschule Zuerich, Institut fuer Informatik

[4] *IDA* Siemens Betriebssystem BS2000, Dialogtesthilfe System 7.700, 2. Ausgabe Januar 1979 (Version 5.0)

[5] *DEBUG:* Siemens ISIS-II 8080/85 Tabellenheft, SME 800 Ausgabe April 1978

[6] J.F. Ossanna *NROFF The Basic Formatting Program* UNIX TIME_SHARING SYSTEM. UNIX Programmers Manual Seventh Edition Volume 1 Bell Telephone Laboratories, Incorporated Murray Hill, New Jersey

NAME

pbug – "screen oriented source level debugger"

SYNOPSIS

pbug <file>

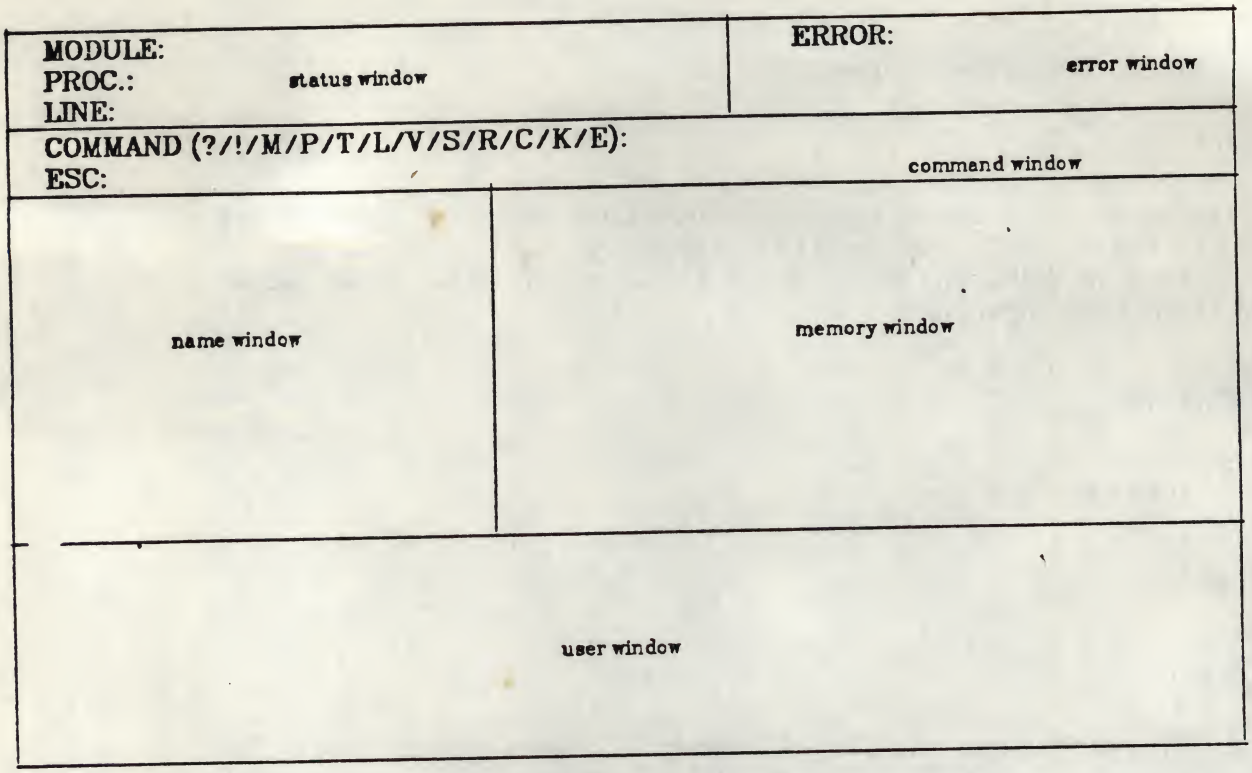
DESCRIPTION

Pbug is an interactive, screen oriented source level debugger. *<file>* is the name of a C- and/or PASCAL program to be debugged.
Pbug is initially in *COMMAND* mode. *P*, *M* etc. (the first letter of the mode) switches to the following modes:

MODULE
PROCEDURE
LINE
VARIABLE
TEMPORARY VARIABLE
SOURCE
RUN
CONTINUE
KILL
END
COMMAND

Hitting *ENTER* switches into the *COMMAND* mode, and gives a 'menu' list of the various commands. '!' and '?' switch off and on 'menu' info, '?' in *COMMAND* mode displays a short description of the modes.

The screen is split into several windows:



STATUS WINDOW: The current state is shown here.

ERROR WINDOW: If an error occurs the messages are shown here. To continue hit the space key.

COMMAND WINDOW: Here you can enter names of variables, files, procedures etc.

NAME WINDOW: Hit ↓ , ↑ or RETURN to come here. Then mark or unmark procedures, variables etc. by hitting '+' or '-'.

MEMORY WINDOW: Memory is displayed here.

USER WINDOW: If your program does terminal I/O, it comes out here.

SOURCE WINDOW: overlays *NAME* , *MEMORY* and *USER WINDOW*s. You can display the source file of a module of your program here and move the window over this file as in *ped (1)*. If you position the cursor to a line containing code, then you can mark or unmark it (if you have compiled this module with the *-g* option (see *cc (1)* or *pc (1)*)).

You can mark or unmark variable names, procedure names etc. (in the following called mark names) in two ways:

Type in '+name' or '-name' in the command window. (*DELETE* or *BACKSPACE* deletes the character left of the cursor position, RETURN ends input).

Go to the *NAME WINDOW* then hit '+' or '-'. The *HOME* key 'amplifies' the following ↓ or ↑ key, e.g. *HOME* ↓ moves the next page of names into the window.

A current name can be defined by typing either 'name' in the *COMMAND WINDOW* or just '>' in the *NAME WINDOW.*

MODULE: the mark name gets the current modul.

PROCEDURE: set or reset a breakpoint at the start ('+name') or end ('-name') of a mark name.

LINE: set or reset a breakpoint at the start of mark name.

VARIABLE: mark or unmark a variable for displaying.

TEMPORARY VARIABLE: mark or unmark a local variable or parameter of the current procedure for displaying.

SOURCE: enter the full pathname of a module of your program to get the source file into the *SOURCE WINDOW.* If the module file is in the current directory. 'S RETURN' looks for current module.c and then for current module.p in the current directory.

RUN: enter the arguments for the subprocess, end with RETURN. Your program will first break at '_entry'.

CONTINUE: hit RETURN to run your program to the next breakpoint, or to the end of the program.

KILL: end the run of your program.

END: end *pbug*.

Before starting *pbug* a special symbol table file is necessary. Run the command *mktree <file>* to create <file>.sym for *pbug*.

When running your program, *pbug* switches to the *USER WINDOW*, so that the program's terminal I/O goes there. If it scrolls the screen up, hit REFRESH to refresh the screen.

If *pbug* stops at a breakpoint, the module and procedure name and the line number are displayed in the *STATUS WINDOW*.
Subcommands here:

'$' display marked variables or temporary variables (switch to 'V' or 'T' mode)
'*' displays all variables or temporary variables

If you hit → in the *NAME WINDOW* at a breakpoint you can change the displayed type of a variable by entering:

'X' 4 bytes hexa
'x' 2 bytes hexa
'I' 4 bytes decimal
'i' 2 bytes decimal
'b' boolean (TRUE/FALSE)

's' 1 byte binary
'c' 1 byte ascii character
'~' 4 bytes hexa (for pointers)

The default is 'X'. Hit 'X' or '~' followed by a space to switch to *MEMORY WINDOW*. Hit space to display the next data memory location in the previous format, or one of the above keys to change the format. '~' dereferences the current memory address. 'd' dumps the following memory locations in 'X' format. Hit ENTER to get back to the *NAME WINDOW*.

KEYBOARD

The CURSOR keys and some other special keys are different from keyboard to keyboard. Therefore the */etc/termcap* data base contains a translation for these keys to an internal representation. See also *termcap (3)* and *termcap (5)* . The new termcap entries all start with 'y'.

y1 => ENTER
y2 => DELETE
y3 => BACKSPACE
y6 => RETURN
y8 => REFRESH
y9 => KILL LINE
yg => CURSOR UP
yh => CURSOR DOWN
yi => CURSOR RIGHT
yj => CURSOR LEFT
yk => CURSOR HOME

FILES

<file>.sym
/etc/termcap

NAME

pc - Pascal compiler

SYNOPSIS

pc [option] ... file ...

DESCRIPTION

pc is the UNIX Pascal compiler. It accepts several types of arguments:

Arguments whose names end with '.p' are taken to be Pascal source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.p'. The '.o' file is normally deleted, however, if a single Pascal program is compiled and loaded all at one go.

The following options are interpreted by pc. See ld(1) for load options.

-c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-d Switch on the debug mode.

-e Suppress extension warning messages.

-n Suppress execution (''dry run''). pc commands.

-o output
Name the final output file output. If this option is used the file 'a.out' will be left undisturbed.

-p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls monitor(3) at the start and arranges to write out a mon.out file at normal termination of execution of the object program. An execution profile can then be generated by use of prof(1).

-w Suppress warning messages.

-Dname=def

-Dname Define the name to the preprocessor, as if by '#define'. If no definition is given, the name is defined as 1.

-Idir Files of '#include' type whose names do not begin

with '/' are always sought first in the directory of the file argument, then in directories named in -I options, then in directories on a standard list.

-L Addtionally generates listings on corresponding files suffixed '.l'.

-P Run only the macro preprocessor and place the result for each '.p' file in a corresponding '.i' file and has no '#' lines in it.

-S Compile the named Pascal programs and leave the assembler-language output on corresponding files suffixed '.s'.

-T Trace and print pc commands. Temporary files are not deleted.

-Uname Remove any initial definition of name.

Other arguments are taken to be either loader(ld) option arguments, or Pascal-compatible object programs, typically produced by an earlier pc run, or perhaps libraries of Pascal-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name a.out.

FILES

file.i	preprocessor output file
file.l	error and listing file
file.o	object file
file.p	input file
file.s	assembler listing file
a.out	loaded (linked) output
/tmp/ptm?	temporaries for pc
/lib/cpp	preprocessor
/lib/pass[12]	compiler for pc
/lib/c2	compiler pass3
/lib/perror	prints errors and listing
/lib/prt0.o	runtime startoff
/lib/mprt0.o	startoff for profiling
/lib/libp.a	pascal runtime-support
/lib/libc.a	standard library, see intro(3)
/usr/include	standard directory for '#include' files

SEE ALSO

K. Jensen and N. Wirth Pascal User Manual and Report
Springer Verlag 1978
monitor(3), prof(1), adb(1), ld(1)

DIAGNOSTICS

The diagnostics produced by pc itself are intended to be self-explanatory. Occasional messages may be produced by the loader(ld).

BUGS

None known, of course.

NAME

ped – "screen oriented editor"

SYNOPSIS

ped <file>

DESCRIPTION

Ped is an interactive, screen oriented editor. *<file>* is the name of the file to be edited. *Ped* is initially in *REPLACE* mode: move the cursor around the screen, then type away. *ENTER i*, *ENTER l* etc. switch to the following modes:

REPLACE
INSERT
LINE
FIND
USE
WINDOW POSIT
SAVE
END EDITING
AREA
PARAGRAPH
TAG
CHANGE KEYS
COMMAND

Cursor position with the four arrow keys: ↓ , ↑ , ← , and → . The *HOME* key 'amplifies' the following cursor key, e.g. *HOME* → moves the cursor to the right margin of the window. If the cursor is already at the right margin, *HOME* → moves the window half a page right. Similarly, *HOME* ↑ moves the cursor to the top margin, or moves up half a page. *HOME HOME* ↓ moves the cursor to the limits of the text, in this case to the last line.

Delete a character: hitting DEL CHAR deletes the character at the cursor position; DELETE or BACK SPACE deletes the character left of the cursor position.

Hitting *ENTER* switches into the *COMMAND* mode, and gives a 'menu' list of the various commands.

LINE operations: delete (d), *insert* (i), *split* (s), *remove* the rest of (r) a line.

FIND: ENTER f find-string ↓ finds the next occurrence of 'find-string' in the file. *ENTER f* ↓ looks for the same 'find-string' again. *ENTER f* ↑ looks up instead of down.

USE: ENTER u <file name> RETURN switches to another file.

WINDOW POSITION: ENTER w line number RETURN or ↓ or ↑ moves to a given line number in the file. *ENTER w +20* ↓ *goes down 20 lines*, *ENTER w -20* ↑ *goes up 20 lines*,
ENTER w +20 → goes right 20 columns.

SAVE, END EDITING asks if you really want to change the file. Answer *y* to save, or *n* to continue editing. *END EDITING* closes the file and goes back to Unix.

In *AREA* mode, first mark a range of lines by:
m -- mark the top line,
move the cursor,
m -- mark the bottom line.
Then you can *delete* (d), *insert* blank lines (i), *copy* (c) the marked lines. *(Copy* copies just below the current cursor position).

PARAGRAPH is like *AREA*, for rectangles instead of line ranges. First:
M -- mark the corner of a rectangle,
move the cursor,
M -- mark the opposite corner of the rectangle.
Then *delete* (d), *horizontal insert* (h), *vertical insert* (v), *replace* (r) with the upper left corner of the rectangle at the current cursor position.

CALLING: If you call *ped* then ped looks for a description file in the directory */usr/lib/red/PE<ttynumber>.<username>* . The contents of such a file is the file name of the last file you edited and the position in that file. If such a file exists for your terminal number and user name, then ped takes the last file you edited and positions where you left it. If you call *ped -f search* or you type *ped -w line* then ped positons in the last file edited to the string *search* or the line *line* .

TAG: If you call *ped -t search* or you type *ENTER t search RETURN* or ↓ or ↑ then ped looks in the file *tags* in the current directory for the entry *search* . (See also *ctags (1))* . It opens the corresponding file and positions to the line where procedure or function *search* is declared. This works for C and PASCAL files.

CHANGE KEYS: to expand a single letter to a long word, like 'M' to 'Mississippi', type:
ENTER ENTER <delimiter> <Mississippi> <delimiter> <M>.
Then hitting 'M' is the same as typing in 'Mississippi'. The expanded 'word' can also contain editor commands, as in:
/ENTER f ↓ENTER r replace/M
The *ENTER x* command displays all expand keys, in the format
<expand key>: <sequence for this key>.

The *ENTER y* command escapes to the unix *shell*. *EOT (control-z)* returns to *Ped*.

KEYBOARD

The CURSOR keys and some other special keys are different from keyboard to keyboard. Therefore the */etc/termcap* data base contains a translation for these keys to an internal representation. See also *termcap (3)* and *termcap (5)* . The new termcap entries all start with 'y'.

y1 => ENTER
y2 => DELETE
y3 => BACKSPACE
y4 => TAB
y5 => DEL CHAR
y6 => RETURN
y7 => BACKTAB
y8 => REFRESH
y9 => KILL LINE
ya => + PAGE
yb => - PAGE
yc => + LINE
yd => - LINE
ye => INSERT LINE
yf => DELETE LINE
yg => CURSOR UP
yh => CURSOR DOWN
yi => CURSOR RIGHT
yj => CURSOR LEFT
yk => CURSOR HOME

If your keyboard has some function keys, then you can install additional functions:
+/- PAGE (LINE) moves the text in the window up/down one (half a) page. INSERT, DELETE LINE inserts or deletes a line without changing the current mode.

FILES

/tmp/text<pid>
/etc/termcap
/usr/lib/red/PE<ttynumber>.<username>

BUGS

AREA is not well tested.

NAME

xref - cross-reference listing

SYNOPSIS

xref [-c] [-p] [-P]

DESCRIPTION

Xref creates a cross-reference listing from the standard input.
It lists on the standard output the alphabetic sorted identifiers followed by the linenumbers in which they appear.

-c C comments will be skipped

-p Pascal comments will be skipped

-P Packed files will be processed. The filenames are prepended to the linenumbers.

Unpacked input files are listed without filenames.

EXAMPLE

pack ack.p fac.p | xref -p -P
outputs:

```
:
var       ack.p     2
writeln   ack.p    12   14                        fac.p     3
x         ack.p     2   13   14   14   15        fac.p    10   12
          fac.p     3   11   12   12   13
y         ack.p     2   13   14   14
:
```

SEE ALSO

pack(1).